图灵程序设计丛书



Peachpit Press

[加] Toby Donaldson 著
袁国忠 译



Python编程入门

（第3版）

Python: Visual QuickStart Guide
Third Edition



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Toby Donaldson

加拿大温哥华西蒙弗雷泽大学计算机科学系高级讲师，教授编程和计算机课程十余年。拥有人工智能博士学位，至今依然醉心于将所学用于他的吸尘机器人。教学之余酷爱鼓捣，拿到什么鼓捣什么。

袁国忠

自由译者。2000年起专事翻译，至今已有14个年头。主译图书，偶译新闻稿、软文。出版译著40余部，其中包括《C++ Prime Plus中文版》、《CCNA学习指南》、《CCNP ROUTE学习指南》、《面向模式的软件架构：资源管理模式》、《风投的选择：谁是下一个十亿美元级公司》等，总计700余万字。专事翻译前，从事过三年化工产品分析和开发工作，做过两年杂志和图书编辑。

图灵程序设计丛书

[加] Toby Donaldson 著
袁国忠 译



Python编程入门

（第3版）

Python: Visual QuickStart Guide
Third Edition

人民邮电出版社
北京



图灵程序设计丛书

Python: Visual QuickStart Guide

Python编程入门（第3版）

[加] Toby Donaldson　著

袁国忠　译

人 民 邮 电 出 版 社

北　　京

图书在版编目（CIP）数据

Python编程入门 : 第3版 / (加) 唐纳森
(Donaldson,T.) 著 ; 袁国忠译. -- 北京 : 人民邮电出
版社, 2013.12
（图灵程序设计丛书）
书名原文: Python:visual quickstart guide,
third edition
ISBN 978-7-115-33374-2

Ⅰ. ①P… Ⅱ. ①唐… ②袁… Ⅲ. ①软件工具－程序
设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2013)第242069号

内 容 提 要

本书是图文并茂的 Python 学习参考书，书中并不包含深奥的理论或者高级应用，而是以大量来自实战的例子、屏幕图和详细的解释，用通俗易懂的语言结合常见任务，对 Python 的各项基础知识进行了介绍，以帮助读者成为一名真正的 Python 程序员。

本书适合各个层次的 Python 开发人员阅读。

◆ 著　　　　[加] Toby Donaldson
译　　　　袁国忠
责任编辑　李　瑛
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：13.25
字数：212千字　　　　2013年 12 月第 1 版
印数：1 – 3 000册　　　　2013年 12 月北京第 1 次印刷

著作权合同登记号　图字：01-2013-6736号

定价：39.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

致　　谢

感谢Clifford Colby和Scout Festa在本书第三版出版过程中贡献的专业知识和耐心；感谢西蒙弗雷泽大学（SFU）的学生们让我明白了学习Python的最佳方式；感谢西蒙弗雷泽大学计算机系的John Edgar和其他教师带给我愉悦的共事体验；感谢Bonnie、Thomas和Emily建议我不要在致谢中喋喋不休。最后要特别感谢Guido van Rossum和Python社区的其他成员，感谢你们打造了一种如此趣味盎然的编程语言。

目　　录

1 编程简介

深入探讨 Python 编程之前，先大致了解一下 Python 是什么及其可用于编写哪些类型的程序，这大有裨益。本章还将概述程序员所做的工作。最后将介绍如何安装 Python 及运行其自带的 IDLE 编辑器。

如果你是编程新手，本章可助你为学习 Python 编程语言做好准备。

如果你已掌握这些基本概念，可跳到有关如何安装 Python 和运行编辑器的章节。

本章内容

- ❑ Python 语言
- ❑ Python 适合用于做什么
- ❑ 程序员如何工作
- ❑ 安装 Python

1.1 Python 语言

那么 Python 是什么呢？简单地说，它是一种计算机编程语言及一组配套的软件工具和库。Python 最初由 Guido van Rossum 于 20 世纪 90 年代初开发，当前由世界各地的数十位程序员（包括 van Rossum）负责维护。

Python 易于理解和学习。相比于用其他大多数编程语言编写的程序，Python 程序更整洁：Python 几乎没有多余的符号，且使用的是简单易懂的英语名称。

Python 语言的效率极高。精通 Python 后，与使用其他大多数编程语言相比，使用 Python 可在更短的时间内完成更多的工作。Python 支持但不强制你使用面向对象编程（OOP）。

Python 自带了各种现成库，供你在自己的程序中使用。有些 Python 程序员喜欢这样说：Python“开箱即可使用”。

Python 的一个极其实用的特点是易于维护。鉴于 Python 程序理解和修改起来相对容易，程序员可轻松地确保它们紧跟潮流。在程序员所做的工作中，程序维护所占的比例很可能高达甚至超过 50%，因此在很多专业人士看来，Python 对维护的支持是个亮点。

最后，说说名称 Python 的由来。据 Python 之父 Guido van Rossum 说，Python 是以喜剧团体 Monty Python（巨蟒小组）的名字命名的。虽然这种起源充满喜庆色彩，但 Python 当前使用的标识确乎是两条缠在一起的蛇（可能是蟒蛇），其中一条为蓝色，另一条为黄色。

1.2　Python 适合用于做什么

虽然 Python 是一种通用语言，可用于编写任何类型的程序，但它最常用于编写下述应用程序。

- **脚本**。这些简短的程序自动执行常见的管理任务，如在系统中新增用户、将文件上传到网站、在不使用浏览器的情况下下载网页等。
- **网站开发**。作为快速创建动态网站的工具，Django（www.djangoproject.com）、Bottle（www.bottlepy.org）和 Zope（www.zope.org）等众多 Python 项目深受开发人员的欢迎。例如，深受欢迎的新闻网站 www.reddit.com 就是使用 Python 开发的。
- **文本处理**。Python 在字符串和文本文件处理方面提供了强大的支持，包括正则表达式和 Unicode。
- **科学计算**。网上有很多卓越的 Python 科学计算库，提供了用于统计、数学计算和绘图的函数。
- **教育**。鉴于 Python 简洁实用，越来越多的学校将其作为第一门编程教学语言。

当然，Python 并非对任何项目来说都是最佳选择，其速度通常比 Java、C#、C++ 等语言慢，因此开发新操作系统时不会使用 Python。

然而，需要最大限度地减少程序员花在项目上的时间时，Python 通常是最佳选择。

1.3 程序员如何工作

虽然对如何编写程序没有严格的规定，但大多数程序员都采用类似的流程。

该程序开发流程如下。

1. 确定程序要做什么，即搞清楚需求。

2. 编写源代码，这里是使用 Python 集成开发环境 IDLE 或其他文本编辑器编写 Python 代码。这一步通常最有趣也最具挑战性，要求你创造性地解决问题。Python 源代码文件使用扩展名 .py，如 web.py、urlexpand.py、clean.py 等。

3. 使用 Python 解释器将源代码转换为目标代码。Python 将目标代码存储在 .pyc 文件中，例如，如果源代码存储在文件 urlexpand.py 中，目标代码将存储在文件 urlexpand.pyc 中。

4. 运行或执行程序。就 Python 而言，通常紧接着第 2 步自动完成这一步。实际上，Python 程序员很少直接与目标代码（.pyc 文件）交互。

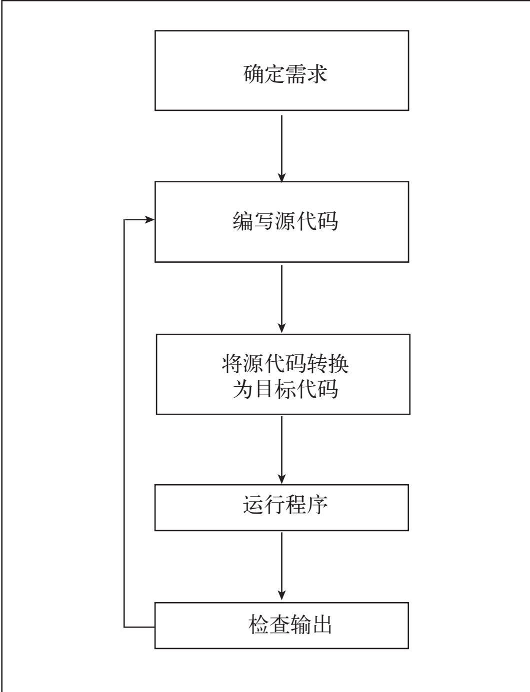


图 1-1 基本计算机程序编写步骤。检查程序输出时，通常会发现错误。为修复错误，必须回到步骤“编写源代码”

术语说明

我们通常将 .py 文件的内容称为程序、源代码或代码。

目标代码有时也称为可执行代码、可执行文件或软件。

5. 最后，检查程序的输出。如果发现错误，回到第 2 步并尽力修复错误。修复错误的过程称为调试。开发庞大或复杂的程序时，可能大部分时间都用在调试上，因此经验丰富的程序员设计程序时，会尽力采用可最大限度地减少调试时间的方式。

如图 1-1 所示，这是个循环往复的过程：编写程序，测试，修复错误，再测试……直到程序正确运行。

1.4 安装 Python

Python 是一种实践性语言，下面来看看如何在计算机上安装它。

1.4.1 在 Windows 系统上安装 Python

步骤如下。

1. 访问 Python 下载页面 www.python.org/download。

2. 选择最新的 Python 3 版本（其名称类似于 Python 3.x，其中 x 是一个较小的数字），这将打开相应的下载页面，其中说明了如何下载用于不同计算机系统的 Python。

3. 根据计算机使用的操作系统，单击相应的安装程序链接。例如，如果是 Windows 操作系统，单击 Windows x86 MSI Installer (3.x)。

4. 下载完毕后，双击安装程序以运行它。

5. 安装完成后（需要几分钟），通过测试看看是否正确安装了 Python。为此，打开“开始”菜单并选择“所有程序”，将看到一个与 Python 3.0 相关的选项（其背景通常为黄色）。选择其中的选项 IDLE (Python GUI)，一段时间后程序 IDLE 将启动，如图 1-2 所示。

6. 输入 24 * 7 并按回车，应出现数字 168。

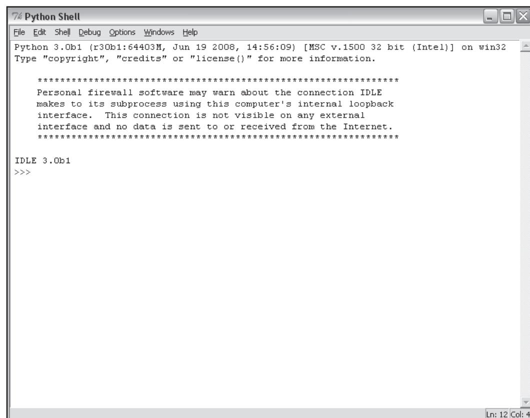


图 1-2 IDLE 编辑器的起始屏幕。第一行指出了当前使用的是哪个 Python 版本（这里为 3.0b1）

1.4.2 在 Mac 系统上安装 Python

OS X 自带并安装了一个 Python 版本，但该版本没有 IDLE 编辑器，通常也不是最新版本。要安装更新的 Python 版本，可按 www.python.org/download/mac/ 给出的说明做，也可从 www.pythonmac.org/packages/ 下载一个安装程序并运行它。下载安装程序时，务必选择正确的 Python 版本（3.0 或更高版本），并确保 Mac OS 版本号与你的操作系统版本号一致。

1.4.3 在 Linux 系统上安装 Python

如果你使用的是 Linux，很可能已经安装了 Python。要确认这一点，可打开命令行窗口并输入 python，如果输出与图 1-2 类似，说明 Python 运行正常。

务必检查版本号。本书介绍的是 Python 3，如果当前安装的是 Python 2.x 或更早的版本，就应安装 Python 3。

具体如何安装因 Linux 系统而异。例如，在 Ubuntu Linux 系统上，需要在 Synaptic Package Manager 中搜索 Python。你也可以访问 www.python.org/download，了解如何在 Linux 系统上安装 Python。

2

算术、字符串与变量

要学习编程，首先要了解基本的 Python 数据类型：整型（整数）、浮点数（带小数点的数字）和字符串。所有程序都使用这些（及其他）数据类型，因此牢固掌握它们的用法至关重要。

使用字符串的程序如此之多，所以 Python 提供了强大的字符串支持。本章将介绍字符串的基本知识，而第 6 章还将回过头来更深入地介绍。

我们还将介绍最重要的编程概念——变量。变量用于存储和操作数据，如果不使用几个变量，就很难编写出有用的程序。

与学习弹钢琴或说外语一样，学习编程的最佳方式也是多练。因此，本章将利用交互式命令 shell（IDLE）来介绍上述所有知识，你最好跟着做：在计算机上输入书里介绍的示例。

本章内容

- 交互式命令 shell
- 整数算术
- 浮点数算术
- 其他数学函数
- 字符串
- 字符串拼接
- 获取帮助
- 类型转换
- 变量和值
- 赋值语句
- 变量如何引用值
- 多重赋值

2.1 交互式命令 shell

我们来看看如何与 Python shell 交互。首先，启动 IDLE。在 Windows 系统中，它位于“开始”菜单的程序列表中；在 Mac 或 Linux 系统中，可直接在命令行输入 python 来启动它。这将打开 Python 交互式命令 shell，它类似于图 2-1。

> **术语说明**
>
> 交互式命令 shell 常简称为交互式 shell、命令 shell、shell 甚至命令行。
>
> shell 记录（transcript）有时称为记录、交互式会话（session）或会话。

2.1.1 shell 提示符

在 Python 记录中，>>> 是 Python shell 提示符。>>> 表示当前行是你（用户）的输入，而没有 >>> 的行是 Python 生成的。因此，一眼就能分辨出哪些内容来自 Python，哪些内容来自用户。

2.1.2 记录

shell 记录是命令行 shell 的快照，显示了一系列用户输入和 Python 的应答。我们将频繁使用 shell 记录。它们让你能够查看实际示例的运行情况，这是学习 Python 的绝佳方式。

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

图 2-1 刚启动 Python 交互式命令 shell 时看到的内容。开头两行指出了你运行的是哪个 Python 版本。这里的信息表明，运行的是 Python 3.3.0，它是 2012 年 9 月 29 日上午 11 点之前的几分钟创建的

2.2　整数算术

整数是不带小数部分的数字，如 25、–86 和 0。Python 支持 4 种基本算术运算：+（加）、–（减）、*（乘）和 /（除）。Python 还使用 ** 和 % 来分别表示乘方和求余，例如，25 % 7 的值为 4，因为 25 除以 7 的余数为 4。下面是一些示例。

```
>>> 5 + 9
14
>>> 22 - 6
16
>>> 12 * 14
168
>>> 22 / 7
3.1428571428571428
>>> 2 ** 4
16
>>> 25 % 7
4
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

2.2.1　整除

Python 还有一个整除运算符 //，其工作原理类似于 /，但结果总是整数。例如，7 // 3 的结果为 2——将小数点后面的值丢弃（而不是四舍五入）。

2.2.2 求值顺序

表 2-1 总结了 Python 的基本算术运算符，并按优先级从低到高的顺序将它们编组。例如，计算表达式 1 + 2 * 3，Python 先执行 *，再执行 +，因为 * 的优先级更高（因此，这个表达式的值为 7，而不是 9）。优先级相同的运算符按书写顺序计算。要改变计算顺序，可使用圆括号 ( )，例如，(1 + 2) * 3 的结果为 9。换句话说，Python 算术运算的规则与常规算术运算相同。

表 2-1　基本算术运算符

名　　称	运　算　符	示　　例
加法	+	>>>3 + 4 7
减法	–	>>> 5 – 3 2
乘法	*	>>> 2 * 3 6
除法	/	>>> 3 / 2 1.5
整除	//	>>> 3 // 2 1
求余	%	>>> 25 % 3 1
乘方	**	>>> 3 ** 3 27

2.2.3 长度不受限制

与其他大多数编程语言不同，Python 对整数的长度没有限制，你可以执行数十位甚至数百数千位的整数运算。

```
>>> 27 ** 100
136891479058588375991326027382088315
→ 966463695625337436471480190078368
→ 997177499076593800206155688941388
→ 250484440597994042813512732765695
→ 774566001
```

2.3 浮点数算术

浮点数算术运行使用的是浮点数。在 Python 中，浮点数是带小数点的数字，例如，–3.1、2.999 和 –4.0 都是浮点数。

所有适用于整数的算术运算都可用于浮点数，包括 %（求余）和 //（整除）。图 2-2 显示了一些示例。

```
>>> 3.8 + -43.2
-39.400000000000006
>>> 12.6 * 0.5
6.3
>>> 12.6 + 0.01
12.61
>>> 365.0 / 12
30.416666666666668
>>> 8.8 ** -5.4
7.939507629591553e-06
>>> 5.6 // 2
2.0
>>> 5.6 % 3.2
2.3999999999999995
```

图 2-2 一些使用 Python 命令 shell 执行基本浮点数算术运算的示例。请注意，近似误差很常见，因此显示的通常不是准确值

2.3.1 浮点数字面量

对于非常大或非常小的浮点数，通常用科学记数法表示。

```
>>> 8.8 ** -5.4
7.939507629591553e-06
```

e-06 表示将它前面的数字乘以 10^{-6}。如果愿意，你可以直接使用科学记数法。

```
>>> 2.3e02
230.0
```

在使用小数点方面，Python 非常灵活。

```
>>> 3.
3.0
>>> 3.0
3.0
```

对于类似于 0.5 的数字，书写时可以包含前导零，也可以不包含。

```
>>> .5
0.5
>>> 0.5
0.5
```

提示 通常 5.0 比 5. 更清晰，因为后者可能令人迷惑，它看起来像句子结尾。

提示 区分 5 和 5.0 很重要，因为 5 是整数，而 5.0 是浮点数，它们的内部表示大相径庭。

2.3.2　溢出

与整数不同，浮点数存在上限和下限，超出上限或下限将导致溢出错误。溢出错误意味着计算结果太大或太小，Python 无法将其表示为浮点数，如图 2-3 所示。面对溢出错误，Python 可能沉默不语，即继续执行错误的计算，而不告诉你出了问题。一般而言，避免溢出错误的职责由程序员承担。

```
>>> 500.0 ** 10000
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    500.0 ** 10000
OverflowError: (34, 'Result too large')
```

图 2-3　浮点数溢出：500.0 ** 10000 的结果太大，无法存储为浮点数

2.3.3　精度有限

无论在哪种计算机上，浮点数的精度都是一个无法解决的难题。在计算机中，数字用二进制（基数为 2）表示，但并非所有浮点数都可用二进制精确地表示。即便在最简单的情况下，也可能出现问题，比如下例。

```
>>> 1 - 2 / 3
0.33333333333333337
```

结果应该是小数点后面有无穷个 3，但这里只包含 17 位。另外，最后一位也不对——应该是 3 而不是 7。

这些细微的误差通常不是问题，对大多数程序来说，小数点后面包含 17 位足够了。然而，当你执行大量计算时，小误差会累积出大误差。例如，计算新设计的桥梁承受的压力时，必须考虑细微的浮点数误差，避免它们累积出大误差。

提 示　一般而言，应优先考虑使用整数而不是浮点数，因为它们更精确且绝不会溢出。

2.3.4　复数

Python 提供了内置的复数支持。复数是涉及 –1 的平方根的数字，在 Python 中，用 `1j` 表示 –1 的平方根。

```
>>> 1j
1j
>>> 1j * 1j
(-1+0j)
```

在有些工程和科学计算中，复数很有用，但本书不会再使用它们。

2.4 其他数学函数

Python 自带了很多由编写好的代码组成的模块，math 就是其中之一。表 2-2 列出了 math 模块中一些最常用的函数。

2.4.1 使用返回值

这些函数返回一个值，这意味着它们的结果为整数或浮点数，具体随函数而异。

在可以使用数字的任何地方，都可使用这些函数。Python 自动执行函数，并将函数调用替换为返回值。

2.4.2 导入模块

要使用模块math或其他任何Python模块，都必须先导入：

```
>>> import math
```

这样就可以访问任何数学函数了，方法是在函数前面加上 math.。

```
>>> math.sqrt(5)
2.23606797749979
>>> math.sqrt(2) * math.tan(22)
0.0125181320236 11912
```

下面是另一种导入模块的方式。

```
>>> from math import *
```

这样调用 math 模块中的任何函数时，都无需在前面加上 math.。

```
>>> log(25 + 5)
3.4011973816621555
>>> sqrt(4) * sqrt(10 * 10)
20.0
```

表 2-2 模块 math 中的一些函数

函　数	描　述
ceil(x)	大于或等于 x 的整数
cos(x)	x 的余弦
degrees(x)	将 x 弧度转换为度数
exp(x)	e 的 x 次方
factorial(n)	计算 n 的阶乘（$n!$）。$n! = 1*2*3\cdots*n$，其中 n 必须是整数
log(x)	以 e 为底的 x 的对数
log(x, b)	以 b 为底的 x 的对数
pow(x, y)	x 的 y 次方
radians(x)	将 x 度转换为弧度数
sin(x)	x 的正弦
sqrt(x)	x 的平方根
tan(x)	x 的正切

提示　使用导入方式 from math import* 时，如果函数与 math 模块中的某个函数同名，将被 math 模块中的同名函数覆盖。

提示　因此，使用导入方式 import math 通常更安全，因为它不会覆盖任何既有函数。

提示　你还可导入模块 math 的特定函数，例如，from math import sqrt, tan 只导入函数 sqrt 和 tan。

2.5 字符串

字符串是一系列字符，如"cat!"、"567-45442"和"Up and Down"。字符包括字母、数字、标点符号以及数百个其他的特殊符号和不可打印的字符。

2

2.5.1 标识字符串

在 Python 中，可使用下列 3 种主要方式来表示字符串字面量。

- 单引号，如 'http'、'openhouse' 或 'cat'。
- 双引号，如 "http"、"open house" 或 "cat"。
- 三引号，如 """http""" 或多行字符串：

```
"""
Me and my monkey
Have something to hide
"""
```

提示 很多 Python 程序员更喜欢使用单引号来标识字符串，这仅仅是因为输入量比使用双引号少（不需要按住 Shift 键）。

提示 单引号和双引号的一个主要用途是，让你能够在字符串中包含字符"和'。

```
"It's great"
'She said "Yes!"'
```

提示 在需要创建多行的长字符串时，三引号很有用。在使用三引号括起的字符串中，还可包含字符"和'。

提示 如果在字符串中包含错误类型的引号，将导致错误。

2.5.2 字符串的长度

要确定字符串包含多少个字符，可使用函数 len(s)，如下所示。

```
>>> len('pear')
4
>>> len('up, up, and away')
16
>>> len("moose")
5
>>> len("")
0
```

最后一个示例使用了空字符串。空字符串通常表示为 '' 或 ""，没有包含任何字符。

由于函数 len 返回一个整数，所以在任何可以使用整数的地方，都可使用函数 len，例如：

```
>>> 5 + len('cat') * len('dog')
14
```

2.6 字符串拼接

可以将既有字符串“相加”来创建新的字符串，比如：

```
>>> 'hot ' + 'dog'
'hot dog'
>>> 'Once' + " " + 'Upon' + ' ' +
→ "a Time"
'Once Upon a Time'
```

这种运算被称为拼接。

要将同一个字符串拼接很多次，可使用下面这种整洁的快捷方式。

```
>>> 10 * 'ha'
'hahahahahahahahahaha'
>>> 'hee' * 3
'heeheehee'
>>> 3 * 'hee' + 2 * "!"
'heeheehee!!'
```

字符串拼接的结果为另一个字符串，因此可在任何需要字符串的地方使用字符串拼接。

```
>>> len(12 * 'pizza pie!')
120
>>> len("house" + 'boat') * '12'
'121212121212121212'
```

2.7 获取帮助

从很大程度上说，Python 是一种自文档化语言，大多数函数和模块都包含简短的解释，你无需求助于图书或网站就能搞明白如何使用它们。

2.7.1 列出模块中的函数

导入模块后，可使用函数 `dir(m)` 列出模块的所有函数。

```
>>> import math
>>> dir(math)
['__doc__', '__name__',
→ '__package__', 'acos', 'acosh',
→ 'asin', 'asinh', 'atan', 'atan2',
→ 'atanh', 'ceil', 'copysign',
→ 'cos', 'cosh', 'degrees', 'e',
→ 'exp', 'fabs', 'factorial',
→ 'floor', 'fmod', 'frexp', 'hypot',
→ 'isinf', 'isnan', 'ldexp', 'log',
→ 'log10', 'log1p', 'modf', 'pi',
→ 'pow', 'radians', 'sin', 'sinh',
→ 'sqrt', 'sum', 'tan', 'tanh',
→ 'trunc']
```

这让你对模块包含的函数有了大致的了解，很多 Python 程序员经常使用 `dir(m)`。

就现在而言，不用考虑以双下划线（`__`）打头的名称，因为它们只用于较复杂的 Python 编程。

提示 要查看完整的 Python 内置函数清单，可在命令提示符下输入 dir(__builtins__)。

提示 你还可使用函数 help(f) 来查看函数 f 的文档字符串。

提示 要运行 Python 实用程序 help，可在提示符下输入 help()。这将向你显示各种有用的信息，如完整的模块清单、有关函数和关键字的帮助信息，等等。

提示 你还可从 Python 文档（www.python.org/doc/）获取帮助。在这里，可找到实用教程以及 Python 语言和标准模块的详情。

2.7.2 打印文档字符串

另一个实用技巧是打印函数的文档字符串。

```
>>> print(math.tanh.__doc__)
tanh(x)
Return the hyperbolic tangent of x.
```

大多数 Python 内置函数都有简短的文档字符串，Python 标准模块（如 math）中的大部分函数亦如此；你可使用前述方式来访问这些文档字符串。

再来看一个例子，下面是内置函数 bin 的文档字符串。

```
>>> print(bin.__doc__)
bin(number) -> string
Return the binary representation of
→ an integer or long integer.
>>> bin(25)
'0b11001'
```

2.8 类型转换

类型转换是一种常见任务，Python 提供了很多简化这种工作的内置函数。

2.8.1 将整数和字符串转换为浮点数

要将整数 3 转换为浮点数，可使用函数 `float(x)`。

```
>>> float(3)
3.0
```

将字符串转换为浮点数的方式与此类似。

```
>>> float('3.2')
3.2000000000000002
>>> float('3')
3.0
```

2.8.2 将整数和浮点数转换为字符串

函数 `str(n)` 将指定的数字转换为相应的字符串。

```
>>> str(85)
'85'
>>> str(-9.78)
'-9.78'
```

隐式转换

有时候 Python 会自动在数值类型之间转换，而不要求你显式地调用转换函数，例如：

```
>>> 25 * 8.5
212.5
```

这里自动将 25 转换为 25.0，再将其与 8.5 相乘。一般而言，表达式同时包含整数和浮点数时，Python 会自动将整数转换为浮点数。

圆整

在 Python 中，round(8.5) 的结果为 8 而不是 9，对此很多人都感到惊讶。你在小学可能学过，对于小数部分为 .5 的数字，总是应该向上圆整的。

然而，总是向上圆整带来的偏差可能导致计算不准确，因此 Python 采用了另一种圆整策略：将小数部分为 .5 的数字圆整到最接近的偶数（有时被称为银行家圆整）。因此，小数部分为 .5 的数字可能向下圆整，也可能向上圆整。

乍一看，这种策略有点奇怪，也不同于 Python 2 的圆整方式。然而，这是在计算机上圆整数字的标准方式，为大家普遍接受。如果你想了解其中的细节，请参阅维基百科的相关词条，其网址为 http://en.wikipedia.org/wiki/Rounding。

2.8.3　将浮点数转换为整数

这有点棘手，因为你必须决定如何处理浮点数的小数部分。函数 int(x) 将小数部分删除，而 round(x) 采用如下标准圆整方式。

```
>>> int(8.64)
8
>>> round(8.64)
9
>>> round(8.5)
8
```

2.8.4　将字符串转换为数字

这很容易，只需使用函数 int(s) 或 float(s) 即可。

```
>>> int('5')
5
>>> float('5.1')
5.1
```

提示　对于大多数应用程序，使用 int(x)、float(x) 和 round(x) 就能满足数值转换需求。然而，为处理更具体的转换，Python 模块 math 提供了很多将小数部分删除的函数：math.trunc(x)、math.ceil(x) 和 math.floor(x)。

提示　函数 int(s) 和 float(s) 将字符串转换为浮点数 / 整数，它们假定字符串看起来像 Python 浮点数 / 整数，如果不是这样，将出现一条错误消息，指出不能执行转换。

2.9 变量和值

变量是最重要的编程概念之一。在 Python 中，变量标记（label）或指向一个值。

例如：

```
>>> fruit = "cherry"
>>> fruit
'cherry'
```

其中，fruit 是一个变量名，它指向字符串值 "cherry"。请注意，变量名无需用引号括起。

代码行 fruit = "cherry" 被称为赋值语句；=（等号）被称为赋值运算符，用于让变量指向一个值。

遇到变量时，Python 将其替换为指向的值，因此：

```
>>> cost = 2.99
>>> 0.1 * cost
0.29900000000000004
>>> 1.06 * cost + 5.99
9.1594000000000015
```

术语说明

与变量一样，函数、模块和类也都有名称。我们将这些名称统称为标识符。

表 2-3　合法和非法的变量名

合法变量名	非法变量名
`M`	`"m"`
`x1`	`1x`
`tax_rate`	`tax rate`
`taxRate`	`taxRate!`
`Else`	`else`

```
>>> else = 25
SyntaxError: invalid syntax
```

图 2-4　else 是 Python 关键字，不能用作变量名

变量命名规则

变量名必须遵守下面几条基本规则（表 2-3 提供了一些示例）。

- 变量名的长度不受限制，但其中的字符必须是字母、数字或下划线（_），而不能使用空格、连字符、标点符号、引号或其他字符。
- 变量名的第一个字符不能是数字，而必须是字母或下划线。
- Python 区分大小写，因此 TAX、Tax 和 tax 是截然不同的变量名。
- 不能将 Python 关键字用作变量名。例如，if、else、while、def、or、and、not、in 和 is 都是 Python 关键字（本书后面将介绍它们的用途），试图将它们用作变量名将导致错误，如图 2-4 所示。

2.10 赋值语句

赋值语句包含 3 个主要部分：左值、赋值运算符和右值，如图 2-5 所示。

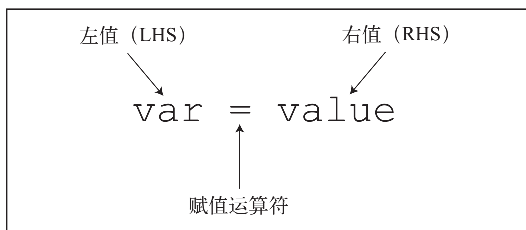


图 2-5 赋值语句剖析。这条语句让 var 指向 value。左值必须是变量，而右值可以是变量、值或结果为值的任何表达式

赋值语句有两个用途：定义新的变量；让已定义的变量指向特定值。例如：

```
>>> x = 5
>>> 2 * x + 1
11
>>> x = 99
```

第一条赋值语句（x = 5）完成了两项职责，是一条初始化语句。它让 Python 创建新变量 x，并将值 5 赋给它。然后，在可以使用整数的任何地方，都可使用变量 x 了。

第二条赋值语句（x = 99）给 x 重新赋值，让它指向另一个值。它没有创建变量 x，因为这个变量已经存在，这是前一条赋值语句的功劳。

如果你不对变量初始化，Python 将报错：

```
>>> 2 * y + 1
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    2 * y + 1
NameError: name 'y' is not defined
```

术语说明

常用于描述变量和值的术语很多。我们有时候说将值赋给变量或给变量指定值。

对于已经赋值的变量，说它指向、标记或拥有相应的值。

程序员有时说变量包含其值，好像变量是桶,而值在桶内。这种说法的问题在于，Python 变量并不符合你以为的“包含”模型。例如，在 Python 中，同一个对象不能同时出现在多个桶内，但可以有多个变量同时指向它。

上述错误消息指出变量 y 未定义，因此 Python 不知道该使用什么值来替换表达式 `2 * y + 1` 中的 y。

可以将任何值赋给变量，包括其他变量的值。请看下面的一系列赋值语句：

```
>>> x = 5
>>> x
5
>>> y = 'cat'
>>> y
'cat'
>>> x = y
>>> x
'cat'
>>> y
'cat'
```

2.11 变量如何引用值

对于 x = expr 这样的 Python 赋值语句，可以这样解读：让 x 指向表达式 expr 的值。

别忘了，expr 可以是任何结果为值的 Python 表达式。

为帮助理解一系列赋值语句，一种不错的方式是绘制示意图。例如，执行语句 rate = 0.04 后，可以认为计算机的内存类似于图 2-6 所示。接下来，执行语句 rate_2008 = 0.06 后，计算机内存类似于图 2-7 所示。最后，执行语句 rate = rate_2008 后，计算机内存类似于图 2-8 所示。

对于没有任何变量指向的值（例如，图 2-8 中的 0.04 ），Python 自动将其删除。一般而言，Python 跟踪所有的值，并自动删除不再有变量指向的值。这称为垃圾收集，因此 Python 程序员很少需要为删除值操心。



图 2-6 执行语句 statement rate = 0.04 后

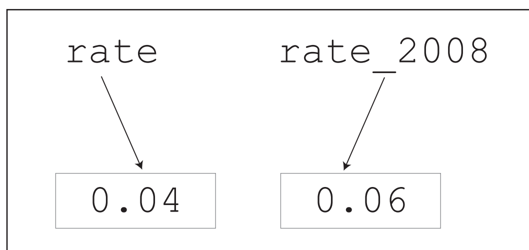


图 2-7 执行语句 rate_2008 = 0.06 后

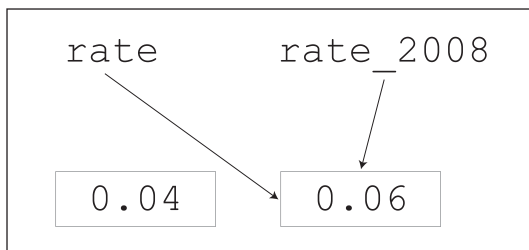


图 2-8 执行语句 rate = rate_2008 后。注意，不再有变量指向 0.04 了，因此 Python 自动将其删除，这个过程称为垃圾收集

2.11.1 赋值时不复制

你必须明白，赋值语句并不会复制指向的值，而只是标记和重新标记既有值。因此，无论变量指向的对象有多大、多复杂，赋值语句的效率都非常高。

2.11.2 数字和字符串是不可变的

在 Python 中，数字和字符串的一个重要特征是不可变，即不能以任何方式修改它们。在看起来是修改数字或字符串的情况下，Python 实际上是在创建修改版本的拷贝，如图 2-9 所示。

```
>>> s = 'apple'
>>> s + 's'
'apples'
>>> s
'apple'
>>> 5 = 1
SyntaxError: can't assign to literal
```

图 2-9 在看起来是修改字符串的情况下，Python 实际上是在创建拷贝。在 Python 中，根本不可能修改数字和字符串

2.12 多重赋值

在 Python 中，有一种便利的技巧，让你能够同时给多个变量赋值：

```
>>> x, y, z = 1, 'two', 3.0
>>> x
1
>>> y
'two'
>>> z
3.0
>>> x, y, z
(1, 'two', 3.0)
```

正如最后一条语句演示的，还可以在一行显示多个值，方法是将它们作为元组。元组总是以左圆括号 (开始,以右圆括号) 结尾。

交换变量的值

多重赋值的一个很实用的用途是交换两个变量的值：

```
>>> a, b = 5, 9
>>> a, b
(5, 9)
>>> a, b = b, a
>>> a, b
(9, 5)
```

语句 a, b = b, a 的含义是，同时给变量 a 和 b 赋值。如果不使用多重赋值，将两个变量的值互换的标准方式如下：

```
>>> a, b = 5, 9
>>> temp = a
>>> a = b
>>> b = temp
>>> a, b
(9, 5)
```

多重赋值的功能并不比常规赋值多，它只是一种偶尔使用的比较便利的快捷方式。

3

编写程序

当目前为止，我们编写的都是单行 Python 语句，并通过交互式命令行运行它们。这对于学习 Python 函数虽然很有用，但当需要编写大量 Python 代码行时，就很烦琐了。

因此，现在开始转而编写程序（也叫脚本）。程序不过是文本文件，但包含一系列 Python 命令。当你运行（或执行）程序时，Python 依次执行文件中的每条语句。

在本章中，你将学习如何在 IDLE 中编写程序，以及如何从 IDLE 和命令行运行程序。你将明白如何获取用户通过键盘提供的输入以及如何将字符串打印到屏幕上。

你应该尽力亲手输入代码，因为这是熟悉各种 Python 编程规则的最佳方式。对于较大的程序，可从本书的配套网站下载代码，网址为 http://pythonintro.googlecode.com。

本章内容

- 使用 IDLE 的编辑器
- 编译源代码
- 从键盘读取字符串
- 在屏幕上打印字符串
- 源代码注释
- 程序的组织

3.1 使用 IDLE 的编辑器

IDLE 自带了一个明白 Python 的文本编辑器。要学习这个编辑器，最佳的方式是编写一个简单程序。

3.1.1 在 IDLE 中编写程序

在 IDLE 中编写程序的步骤如下。

1. 启动 IDLE。

2. 选择菜单 File > New Window。

3. 输入下面的代码：

```
print('Welcome to Python!')
```

4. 选择菜单 File > Save 将程序存盘。将其存储在你的 Python 程序文件夹中，并命名为 `welcome.py`；末尾的 `.py` 表明这是一个 Python 文件。

5. 选择菜单 Run > Run Module 运行程序。

将出现一个 Python shell，其中显示了“Welcome to Python!”。

熟悉 IDLE 编辑后，你可能想使用表 3-1 列出的一些快捷键，它们确实能提高编辑速度。

> **提 示** 在计算机桌面上创建一个特殊文件夹，并将其命名为 python，用于存储你的所有 Python 程序。不要将它们存储到 Python 安装目录中，否则将面临无意间覆盖 Python 核心文件的风险。

表 3-1 一些实用的 IDLE 快捷键

命 令	作 用
Ctrl-N	打开一个新的编辑器窗口
Ctrl-O	打开一个文件进行编辑
Ctrl-S	保存当前程序
F5	运行当前程序
Ctrl-Z	撤销最后一次操作
Shift-Ctrl-Z	重做最后一次操作

> **提 示** 你必须按原样输入 Python 程序，一个字符都不差。哪怕错一个字符(多一个空格、将数字 1 错误地输入为字母 l)，都可能导致错误。

> **提 示** 如果你运行程序时看到错误消息，请返回到编辑器窗口，并逐字符地仔细核对程序，确保输入正确。

其他编辑器

对初学者来说，IDLE 是一款杰出的编辑器，连一些专业人员都经常使用它。然而，如果你不喜欢 IDLE，可在网上使用 programming editors（编程编辑器）进行搜索，你将得到大量的建议。例如，在 Windows 系统上，Notepad++ 是一款深受欢迎的编程编辑器，而且是免费的。另一款深受欢迎的编辑器是 Sublime Text，它有 Windows、Mac 和 Linux 版本，但不是免费的。

要获悉众多其他建议，请参阅 http://wiki.python.org/moin/PythonEditors。

3.1.2 从命令行运行程序

另一种运行 Python 程序的常见方式是，从命令行运行。例如，要运行 welcome.py，可以打开命令行窗口，并执行下述命令：

```
C:\> python welcome.py
Welcome to Python!
```

也可以只调用 Python，而不指定程序，这将打开交互式解释器的缩简版本，但依然很有用。

3.1.3 从命令行调用 Python

执行如下命令：

```
C:\> python
Python 3.0b2 (r30b2:65106, Jul 18 2008,
→ 18:44:17) [MSC v.1500 32 bit (Intel)]
→ on win32
Type "help", "copyright", "credits" or
→ "license" for more information.
>>>
```

将 Python 脚本作为其他程序的一部分进行运行时，通常从命令行调用 Python。

提 示 在 Windows 系统中，打开命令行窗口的最简单方式是，单击“开始”按钮，在“运行”对话框中输入 cmd 并按回车键。这将打开一个命令行窗口。

提 示 在 Mac 和 Linux 系统中，从命令行运行 Python 程序的方式相类似：打开命令 shell（具体如何打开随系统而异，但你可通过桌面上的菜单浏览可用的程序），再输入 python 以及要运行的程序的名称。

提 示 从命令行运行 Python 时，令人讨厌的一点是通常需要配置环境变量（具体地说是系统的路径变量），让系统知道到计算机的什么地方去寻找 Python。关于具体如何配置的介绍过于烦琐且随系统而异，这超出了本书的范围。然而，如果你要配置，很容易在网上找到详细的说明。例如，只需在喜欢的搜索引擎中输入 set windows path 即可。修改环境变量时务必小心：如果你拿不准该如何做，很可能破坏系统，导致程序再不能正确运行。在这种情况下，最佳的选择是从头再来并重新安装 Python。

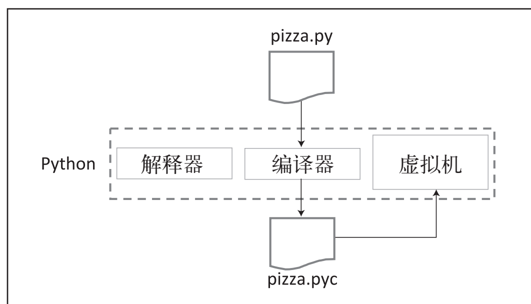


图 3-1　Python 由 3 个主要部分组成：运行语句的解释器、将 .py 文件转换为 .pyc 文件的编译器以及运行 .pyc 文件的虚拟机。请注意，严格地说 IDLE 并非 Python 的一部分，它是一个位于 Python 之上的独立应用程序，旨在让 Python 使用起来更容易

3.2　编译源代码

我们经常将 Python 程序中的语句称为源代码，并将程序文件称为源代码文件。根据约定，所有 Python 源代码文件都使用扩展名 .py。这让人和程序一眼就能明白文件包含 Python 源代码。

3

目标代码

当运行 .py 文件时，Python 会自动创建相应的 .pyc 文件，如图 3-1 所示。.pyc 文件包含目标代码（编译后的代码）。目标代码基本上是一种 Python 专用的语言，以计算机能够高效运行的方式表示 Python 源代码。这种代码并不是供人类阅读的，因此在大多数情况下你都应对 .pyc 文件置之不理。

Python 程序是使用名为虚拟机的特殊软件运行的。这个软件模拟计算机，是专为运行在 Python 上而设计的，这让很多 .pyc 文件无需做任何修改就能在不同的计算机系统上运行。

提 示　你几乎不用关心 .pyc 文件。Python 在需要时会自动创建它们，并在你修改了相应的 .py 文件时自动更新它们。千万不要删除、重命名或修改 .pyc 文件！

提 示　鉴于 .pyc 文件仅供计算机阅读，因此它们不是文本文件。如果你试图在文本编辑器中查看 .pyc 文件，看到的将是一堆乱码。

3.3 从键盘读取字符串

从键盘读取字符串是从用户那里获取信息的一种最基本的方式。例如，请看下面这个简单程序：

```
# name.py
name = input('What is your first name? ')
print('Hello ' + name.capitalize()+ '!')
```

要在 IDLE 中运行它，请在 IDLE 窗口中打开 name.py，再按 F5（或选择菜单 Run > Run Module）。此时将出现一个窗口：

```
What is your first name? jack
Hello Jack!
```

你（用户）必须输入名字（这里为 jack）。

3.3.1 跟踪程序

下面来仔细研究这个程序的每一行。第 1 行是源代码注释（简称为注释）。注释不过是给程序员阅读的说明，Python 对其置之不理。Python 注释总是以符号 # 打头，并延续到行尾。这里的注释指出，这个程序存储在文件 name.py 中。

第 2 行调用函数 `input`，这是用于从键盘读取字符串的标准内置函数。这行代码执行时，将在输出窗口中显示 `What is your first name?` 和闪烁的光标。程序等待用户输入一个字符串并按回车。

函数 input 返回用户输入的字符串，让变量 name 最终指向用户输入的字符串。

该程序的第 3 行（也是最后一行）显示一句问候语。函数 name.capitalize() 确保字符串的第一个字符为大写，而其他字符为小写。这样，如果用户输入的名字没有采用正确的首字母大写方式，Python 将更正。

3

提 示 要获悉字符串包含哪些函数，可在 IDLE 的交互式命令行输入 dir('')。

提 示 如果你多次运行程序 name.py，且每次都输入不同的字符串，将很快发现当你输入类似于 'Jack Aubrey' 的姓名时，姓的首字母将被转换为小写：'Hello Jack aubrey!'。这是因为函数 capitalize 的头脑非常简单，根本没有单词和空格的概念。

提 示 从键盘读取字符串时，另一种常见的实用技巧是，使用函数 strip() 将开头和末尾的空白字符删除，如下所示：

```
>>> ' oven '.strip()
'oven'
```

因为经常需要删除不需要的空白，所以我们常常像下面这样调用函数 input：

```
name = input('Enter age: ').strip()
```

3.3.2 从键盘读取数字

函数 input 只是返回字符串，因此如果你需要的是数字（如用于算术运算），就必须使用 Python 的数值转换函数之一。例如，请看下面的程序：

```
# age.py
age = input('How old are you today? ')
age10 = int(age) + 10
print('In 10 years you will be ' +
→ str(age10) + ' years old.')
```

假设运行该程序时用户输入 22，变量 age 将指向字符串 '22'，因为 Python 不会自动将看起来像数字的字符串转换为整数或浮点数。如果你要将字符串用于算术运算，必须先将其转换为数字。为此，可使用函数 int(s)（如果你需要的是整数）或 float(s)（如果你需要的是浮点数）。

这里要指出的最后一个技巧是，在 print 语句中，必须将变量 age10（它指向一个整数）转换为字符串，这样才能打印它。如果你忘记这样做，Python 将显示错误消息，指出不能将数字与字符串相加。

不同类型的数字

一开始，各种不同的数值类型令你迷惑。请看下面 4 个不同的值：5、5.0、'5' 和 '5.0'。虽然它们看起来相似，但内部表示截然不同。

5 是一个整数，可直接用于算术运算。

5.0 是一个浮点数，也可用于算术运算，但包含小数部分。

'5' 和 '5.0' 都是字符串，分别包含 1 个和 3 个字符。字符串可显示到屏幕上或用于基于字符的操作（如删除空白或计算字符数）。字符串不能用于算术运算。当然，字符串可用于拼接，虽然结果可能让人觉得有点不合情理。例如：

```
>>> 3 * '5'
'555'
>>> 3 * '5.0'
'5.05.05.0'
```

> **术语说明**
>
> 程序员常常使用术语标准输出（stdout）来表示文本被打印到的窗口。通常，标准输出是简单的文本窗口，几乎只显示字符串，不显示任何类型的图形。
>
> 同样，标准输入（stdin）是函数 input 从中读取字符串的地方，通常是标准输出对应的窗口，但必要时可更改标准输入和标准输出。
>
> 你有时还会看到输出标准错误（stderr），它指的是显示错误消息的地方。默认情况下，错误消息通常在标准输出中显示。

3.4 在屏幕上打印字符串

print 语句是用于将字符串打印到屏幕的标准内置函数。正如你将看到的，它非常灵活，有很多功能可用于正确地设置字符串和数字的格式。

你可将任意数量的字符串传递给 print：

```
>>> print('jack', 'ate', 'no', 'fat')
jack ate no fat
```

默认情况下，print 在标准输出窗口中打印每个字符串，并用空格分隔它们。修改字符串分隔符很容易，可以像下面这样做：

```
>>> print('jack', 'ate', 'no', 'fat',
→ sep = '.')
jack.ate.no.fat
```

默认情况下，print 打印完指定内容后添加一个换行符：\n。换行符导致光标移到下一行，因此默认情况下，调用 print 后不能在同一行打印任何内容：

```
# jack1.py
print('jack ate ')
print('no fat')
```

上述代码打印两行文本：

```
jack ate
no fat
```

要在同一行打印所有文本，可将第一行的结束字符指定为空字符串：

```
# jack2.py
print('jack ate ', end = '')
print('no fat')
```

提示 Python 2 和 Python 3 的主要差别之一就表现在函数 print 上。在 Python 2 中，print 从技术上说并非函数，而是语言的一部分。这带来的优点之一是，你可以不输入圆括号，例如，可以输入 print 'jack ate no fat'。然而，print 不是函数虽然带来了这种小小的便利，但导致修改默认分隔符和结束字符串非常困难，而在比较复杂的程序中常常需要这样做。

提示 Python 2 和 Python 3 的另一个不同之处在于，Python 3 函数 input 对应于 Python 2 函数 raw_input；Python 2 也有函数 input，但它对用户输入的字符串求值，这在有些情况下非常方便。在 Python 3 中，没有与 Python 2 函数 input 等价的函数，但可轻松地模拟这个函数——使用 eval(input(prompt))。例如：

```
>>> eval(input('? '))
? 4 + 5 * 6
34
```

3.5 源代码注释

前面使用了源代码注释来指出文件名，但注释还可用于在程序中添加各种说明，如文档、提示、解释或警告。Python 忽略所有注释，它们仅供你和其他可能阅读源代码的程序员阅读。

3

下面的示例程序演示了注释的其他一些用途：

```
# coins_short.py
# This program asks the user how many
# coins of various types they have,
# and then prints the total amount
# of money in pennies.
# get the number of nickels, dimes,
# and quarters from the user
n = int(input('Nickels? '))
d = int(input('Dimes? '))
q = int(input('Quarters? '))
# calculate the total amount of money
total = 5 * n + 10 * d + 25 * q
# print the results
print() # prints a blank line
print(str(total) + ' cents')
```

3.6　程序的组织

随着编写的程序越来越多，你将很快发现它们通常采用相同的结构。人们通常按图 3-2 所示的方式组织程序：包含输入部分、处理部分和输出部分。

在我们刚开始编写的小型程序中，这种结构通常显而易见，无需做太多的考虑。但随着程序越来越大、越来越复杂，很容易偏离这种总体结构，其结果常常是代码混乱、难以理解。

因此，应该养成良好的习惯——使用注释指出输入、处理和输出部分。这有助于阐明程序执行的不同任务。当你开始编写函数时，将发现这种结构提供了很好的指导，让你能够将程序合理地划分成多个函数。

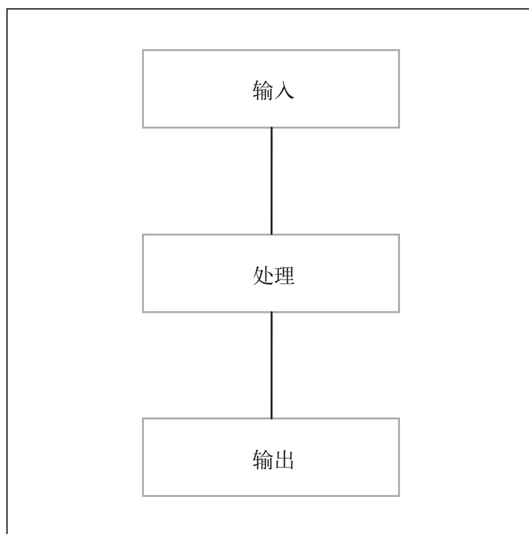


图 3-2　大多数程序都采用这里所示的结构：首先获取输入（例如，使用函数 input 从用户那里获取），然后对输入进行处理，最后向用户显示结果

4

流程控制

前面编写的程序都是直线式的，由一系列依次执行的 Python 语句组成。这些程序按顺序执行语句，没有分支，也不会返回到以前的语句。

本章介绍如何使用 if 语句和循环来改变语句的执行顺序。对任何重要的程序来说，if 语句和循环都是必不可少的。

if 语句和循环都由逻辑表达式控制，因此本章将首先介绍布尔逻辑。

请仔细阅读本章的示例程序，并花点时间进行尝试和修改。

本章内容

- ❑ 布尔逻辑
- ❑ if 语句
- ❑ 代码块和缩进
- ❑ 循环
- ❑ 比较 for 循环和 while 循环
- ❑ 跳出循环和语句块
- ❑ 循环中的循环

4.1 布尔逻辑

与大多数编程语言一样，Python 也使用布尔逻辑来做决策。布尔逻辑就是操作真值，而在 Python 中，这些真值用 `True` 和 `False` 表示。布尔逻辑比算术运算简单，对你知道的逻辑规则进行了规范化。

我们使用 4 个主要的逻辑运算符（也叫逻辑连接符）来组合布尔值：`not`、`and`、`or` 和 `==`。在 Python 及所有计算机语言中，所有决策都可使用这些逻辑运算符来做出。

假设 `p` 和 `q` 是两个 Python 变量，且都是布尔值。由于 `p` 和 `q` 都有两个可能取值（`True` 或 `False`），所以有 4 种不同的组合，如表 4-1 的前两列所示。现在可以这样定义逻辑运算符了:用逻辑运算符将真值 `p` 和 `q` 连接起来时，得到的结果是什么。这些定义被称为真值表，而 Python 使用它们的内部版本来计算布尔表达式的值。

表 4-1　基本逻辑运算符的真值表

p	**q**	**p == q**	**p != q**	**p and q**	**p or q**	**not p**
False	False	True	False	False	False	True
False	True	False	True	False	True	True
True	False	False	True	False	True	False
True	True	True	False	True	True	False

4.1.1 逻辑相等

咱们从 == 开始吧。仅当 p 和 q 包含的真值相同，即都为 True 或都为 False 时，表达式 p == q 的结果才为 True。表达式 p != q 检测 p 和 q 是否不同，仅在 p 和 q 不同时才返回 True。

```
>>> False == False
True
>>> True == False
False
>>> True == True
True
>>> False != False
False
>>> True != False
True
>>> True != True
False
```

4.1.2 逻辑与

仅当 p 和 q 都为 True 时，布尔表达式的结果才为 True，而在其他情况下都为 False。表 4-1 的第 5 列总结了各种组合的结果。

```
>>> False and False
False
>>> False and True
False
>>> True and False
False
>>> True and True
True
```

4.1.3 逻辑或

仅当 p 和 q 至少有一个为 True 时，布尔表达式 p or q 才为 True。表 4-1 的第 6 列对此做了总结。唯一一个稍微有点棘手的情形是，p 和 q 都为 True。在这种情形下，表达式 p or q 的结果为 True。

```
>>> False or False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
```

4.1.4 逻辑非

最后，在 p 为 False 时，布尔表达式 not p 的结果为 True；而在 p 为 True 时，结果为 False。结果与变量的值相反。

```
>>> not True
False
>>> not False
True
```

4.1.5 计算较长的布尔表达式

布尔表达式用于控制 if 语句和循环，你必须明白如何计算它们的值，这很重要。与算术表达式一样，布尔表达式使用圆括号和运算符优先级来指定其各个部分的计算顺序。

4.1.6 计算包含圆括号的布尔表达式

假设我们要计算表达式 not (True and (False or True)) 的值,可以按下面的步骤做。

1. 总是首先计算圆括号内的表达式，因此首先计算 False or True，结果为 True。因此，原来的表达式与下面这个更简单的表达式等价：

```
not (True and True)
```

2. 为计算这个表达式，我们再次先计算圆括号内的表达式：True and True，其结果为 True。因此，上述表达式等价于表达式 not True。

3. 最后，为计算这个表达式，我们只需在表 4-1 的最后一列查找答案：not True 的结果为 False。因此表达式 not (True and (False or True)) 的结果为 False。要使用 Python 来验证这一点很容易，只需像下面这样做：

```
>>> not (True and (False or True))
False
```

表 4-2　布尔运算的优先级（按从高到低的顺序排列）

p == q
p != q
not p
p and q
p or q

4.1.7　计算不包含圆括号的布尔表达式

假设要计算表达式 not True and False orTrue 的值。这个表达式与前一个相同，但没有圆括号。

1. 首先计算优先级最高的运算（表 4-2 列出了运算符的优先级）。在这里，not 的优先级最高，因此首先计算 not True（它位于表达式开头，但纯属巧合）。经过这种计算后，整个表达式简化成了 False and False or True。

2. 再次计算优先级最高的运算符。根据表 4-2 可知，and 的优先级比 or 高，因此先计算 False and False。表达式简化为 False or True。

3. 通过查找表 4-1 中的答案可知，这个表达式的结果为 True。因此，最初的表达式 False and not False or True 的结果为 True。

提 示　编写复杂的布尔表达式时，不使用圆括号通常是个馊主意，因为这将导致表达式难以阅读和计算——并非所有程序员都能记住布尔运算符的优先级顺序！

提 示　一个例外是连续多次使用同一个逻辑运算符时。在这种情况下，不使用圆括号通常更容易阅读，例如：

```
>>> (True or (False or (True or False)))
True
>>> True or False or True or False
True
```

4.1.8 短路求值

表 4-1 所示的逻辑运算符定义是标准定义，为所有逻辑教科书所采用。然而，与大多数现代编程语言一样，Python 也采用了一个被称为短路求值的技巧，这可提高某些布尔表达式的计算速度。

请看布尔表达式 `False and X`，其中 X 为任何布尔表达式。不管 X 的结果为 True 还是 False，整个表达式的结果都为 False，因为开头的 False 导致整个 and 表达式为 False，所以表达式 `False and X` 的值不依赖于 X——总是为 False。在这样的情况下，Python 根本不计算 X 的值，而在遇到 `False and` 时就此罢手，并返回结果 False。这可提高布尔表达式的计算速度。

同样，布尔表达式 `True or X` 的结果总是为 True，而不管 X 的值是什么。表 4-3 描述了 Python 进行短路求值时遵循的规则。

在大多数情况下，你都无需考虑短路问题，就能享受它带来的性能改善。然而，牢记 Python 进行短路求值很有帮助，因为它偶尔会导致难以发现的 bug。

提 示 利用表 4-3 中 and 和 or 的定义，可编写简洁而巧妙的代码来模拟下一节将介绍的 if 语句。然而，这种表达式的可读性通常极差，如果你在他人编写的 Python 代码中遇到这种表达式（在你自己的程序中，绝不要这样做），可能需要参考表 4-3 来搞清楚他们想做什么。

表 4-3 Python 中的布尔运算符定义

运　算	结　果
`p or q`	如果 p 为 False，结果为 q，否则结果为 p
`p and q`	如果 p 为 False，结果为 p，否则结果为 q

4.2 if 语句

if 语句让你能够改变 Python 程序的执行流程，它能让你在程序中做出决策：在程序运行时，该运行这个代码块还是另一个代码块。几乎所有重要的程序都使用 if 语句，因此你必须明白这种语句。

if/else 语句

假设你要编写一个检查密码的程序：用户输入密码，如果正确，就让用户使用其账户登录；否则告诉用户输入的密码不正确：

```
# password1.py
pwd = input('What is the password? ')
if pwd == 'apple': # note use of ==
➝ # instead of =
  print('Logging on ...')
else:
  print('Incorrect password.')
print('All done!')
```

这个程序很容易理解：如果变量 pwd 指向的字符串为 'apple'，就打印一条登录消息；否则打印消息 Incorrect password。

if 语句总是以关键字 if 打头，而这个关键字后面总是一个布尔表达式。这种表达式被称为 if 条件（简称为条件）。if 条件后面是一个冒号（:）。正如你将看到的，Python 将 : 用作 if 语句头、循环头和函数头的结束标记。

从 if 到 : 的部分被称为 if 语句头。如果 if 语句头中的条件为 True，将立即执行语句 print('Logging on ...')，并跳过语句 print('Incorrect password.')，永远不执行它。

如果 if 语句头中的条件为 False，将跳过语句 print('Logging on ...')，只执行语句 print('Incorrect password.')。无论在哪种情况下，都会执行最后的语句 print('All done!')。

图 4-1 说明了 if/else 语句的基本结构。

提示　我们通常将包含多行的整个 if 结构是为一条 if 语句。

提示　你必须在关键字 if 后面至少添加一个空格。

提示　关键字 if、条件和结尾的 : 必须位于同一行。

提示　在 if 语句中，else 代码块是可选的，可根据要解决的问题决定是否包含它。

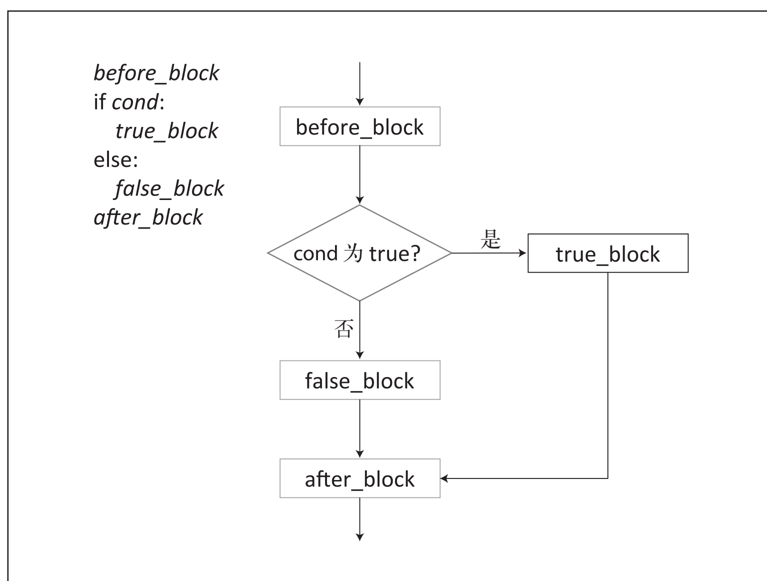


图 4-1　这个流程图说明了 if/else 语句的基本格式和行为，其中的代码块可包含任意数量的 Python 语句（包括其他 if 语句！）

4.3 代码块和缩进

Python 的一个与众不同之处是，使用缩进来标识代码块。请看下述 if 语句，它摘自前述检查密码的程序：

```
if pwd == 'apple':
  print('Logging on ...')
else:
  print('Incorrect password.')
print('All done!')
```

代码行 print('Logging on ...') 和 print ('Incorrect password.') 是两个不同的代码块。这些代码块只有一行，但 Python 允许你编写包含任意数量语句的代码块。

要在 Python 中标识代码块，必须以同样程度缩进代码块的每一行。在前述 if 语句示例中，两个代码块都缩进了 4 个空格，这是典型的 Python 缩进量。

在其他大多数编程语言中，缩进只用于让代码更美观；但在 Python 中，必须使用缩进来指出语句所属的代码块。例如，最后一条语句 final print('All done!') 没有缩进，因此不属于 else 代码块。

对于缩进很重要的理念，熟悉其他语言的程序员常常不以为然：很多程序员喜欢自由，希望能够随心所欲地设置代码的格式。然而，Python 的缩进规则符合很多程序员用来改善代码可读性的风格。Python 只是让这种理念更进一步，赋予缩进以一定的含义。

提示 IDLE 自动为你缩进代码。例如，当你在 if 语句头中输入：并按回车键时，将自动在下一行缩进光标。

提示 缩进量很重要，在 Python 语句块中，多一个或少一个空格都可能导致错误或意外行为。在同一个代码块中，所有语句的缩进量必须相同。

4.3.1 if/elif 语句

if/elif 语句是 if 语句推广版本，它包含多个条件，用于做出复杂的决策。例如，假设航空公司提供了如下儿童优惠票价：不超过 2 岁的儿童免费；2~13 岁的儿童打折；13 岁及更大的儿童与成人同价。下面的程序计算乘客应支付的费用：

```
# airfare.py
age = int(input('How old are you? '))
if age <= 2:
  print(' free')
elif 2 < age < 13:
  print(' child fare)
else:
  print('adult fare')
```

从用户那里获取 age 的值后，Python 进入 if/elif 语句，按指定顺序依次检查每个条件。因此，它首先检查 age 是否小于 2，如果是这样，就指出免费并跳出 if/elif 语句。如果 age 不小于 2，则检查下一个 elif 条件，看看 age 是否在 2 和 13 之间。如果是这样，就打印相应的消息，并跳出 if/elif 语句。如果 if 条件和 elif 条件都不为 True，则执行 else 代码块中的语句。

提示 elif 是 else if 的缩写，你可根据需要使用任意数量的 elif 语句块。

提示 在 if/elif 语句中，每个代码块的缩进量必须相同。

提示 与常规 if 语句一样，else 代码块是可选的。在包含 else 语句块的 if/elif 语句中，只有一个语句块会被执行；如果没有 else 语句块，则可能没有任何条件为 True，在这种情况下，将不会执行 if/elif 语句中的任何代码块。

4.3.2　条件表达式

Python 还有一个逻辑运算符，但有些程序员喜欢，有些程序员讨厌！它基本上就是 if 语句的简写，可直接用于表达式中。请看下面的代码：

```
food = input("What's your favorite food? ")
reply = 'yuck' if food == 'lamb' else 'yum'
```

在第 2 行，= 右边的表达式被称为条件表达式，其结果要么为 'yuck' 要么为 'yum'。上述代码与下面的代码等价：

```
food = input("What's your favorite food? ")
if food == 'lamb':
  reply = 'yuck'
else:
  reply = 'yum'
```

条件表达式通常比等价的 if/else 语句简短，但并非总是与 if/else 语句一样灵活和易于理解。一般而言，仅当条件表达式可让代码更简单时才使用它们。

4.4 循环

下面将注意力转向循环。循环用于重复地执行代码块，Python 有两种主要的循环：for 循环和 while 循环。for 循环通常比 while 循环更容易使用，也不那么容易出错，但没有 while 循环灵活。

4.4.1 for 循环

基本 for 循环重复执行给定代码块指定的次数。例如，下面的代码片段在屏幕上打印数字 0~9：

```
# count10.py
for i in range(10):
  print(i)
```

for 循环的第 1 行被称为 for 循环头。for 循环总是以关键字 for 打头，接下来是循环变量（这里为 i），然后是关键字 in。关键字 in 后面通常（但并非总是）是 range(n) 和结束符号 :。for 循环重复执行循环体（循环头后面的语句块）*n* 次。

每次执行循环时，循环变量都被设置为下一个值。默认情况下，初始值为 0，并逐步递增到 *n*–1（而不是 *n*）。从 0 而不是 1 开始可能看起来有点怪，但在编程中很常见。如果要修改循环的初始值，可在 range 中添加初始值：

```
for i in range(5, 10):
  print(i)
```

上述代码打印数字 5~9。

术语说明

程序员经常使用变量 i，因为它是索引（index）的缩写，且常用于数学中。等你开始使用循环中的循环时，通常将 j 和 k 用作其他循环变量名。

提示　如果你要打印数字 1~10（而不是 0~9），有两种常见的办法，一种是修改范围的起始值和终止值。

```
for i in range(1, 11):
  print(i)
```

另一种是在循环体中给 i 加 1。

```
for i in range(10):
  print(i + 1)
```

提示　如果你想按相反的顺序打印数字，也有两种标准做法。第一种是这样设置范围参数。

```
for i in range(10, 0, -1):
  print(i)
```

注意到第一个参数为 10，第二个参数为 0，而第三个参数（被称为步长）为 –1。另一种做法是使用更简单的范围，并在循环体中修改 i。

```
for i in range(10):
  print(10 - i)
```

提示　for 循环实际上比本节描述得更通用。可将 for 循环用于任何类型的迭代器——一种返回值的特殊编程对象。例如，for 循环是读取文本文件各行的最简单方式，你将本书在后面看到这一点。

4.4.2 while 循环

第二种 Python 循环是 while 循环。请看下面的程序：

```
# while10.py
i = 0
while i < 10:
  print(i)
  i = i + 1 # add 1 to i
```

这个程序在屏幕上打印数字 0~9。它比 for 循环要复杂得多，但也更灵活。

while 循环本身以关键字 while 打头，这一行被称为 while 循环头，而它后面缩进的代码被称为 while 循环体。

while 循环头总是以 while 打头，后面跟着循环条件——返回 True 或 False 的布尔表达式。

while 循环的控制流程类似于这样：首先，Python 检查循环条件为 True 还是 False，如果为 True，就执行循环体；如果为 False，就跳过循环体（即跳出循环）并执行后面的语句。在条件为 True 并执行循环体后，Python 再次检查条件。只要循环条件为 True，Python 就不断执行循环。图 4-2 是这个程序的流程图。

示例程序的第 1 行为 `i = 0`，在循环语境下，这种语句被称为*初始化语句*。与 for 循环那样自动初始化循环变量不同，由程序员负责给 while 循环使用的变量指定初始值。

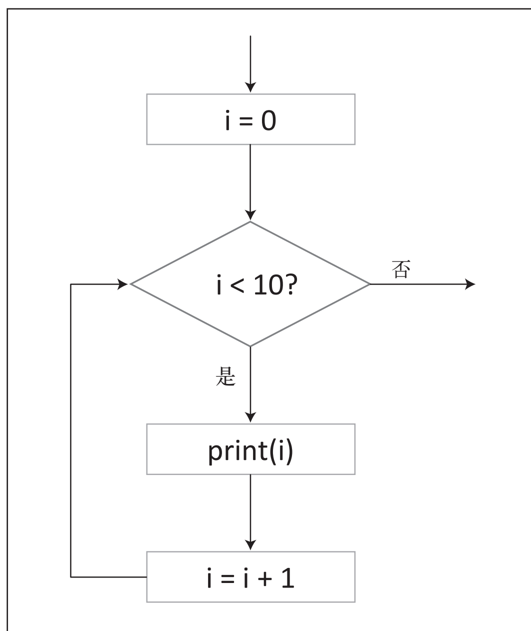


图 4-2　这是从 0 数到 9 的代码对应的流程图。注意到循环条件为 False（即从决策箱进入“否”分支）时，箭头并没有指向一个箱子，这是因为在这里的示例代码中，while 循环后面没有任何代码

循环体的最后一行为 `i = i + 1`。正如源代码注释指出的，这行代码将 i 的值加 1。因此，随着循环的进行，i 的值不断递增，确保循环终将终止。在 while 循环中，这一行被称为递增语句，因为其职责是让循环变量递增。

图 4-3 所示的流程图演示了通用的 while 循环。

虽然几乎所有 while 循环都需要初始化语句和递增语句，但 Python 并未要求必须包含它们。确保包含这些代码行完全是程序员的职责。即便是经验丰富的程序员也会发现，初始化语句和递增语句是导致错误的常见罪魁祸首。

4

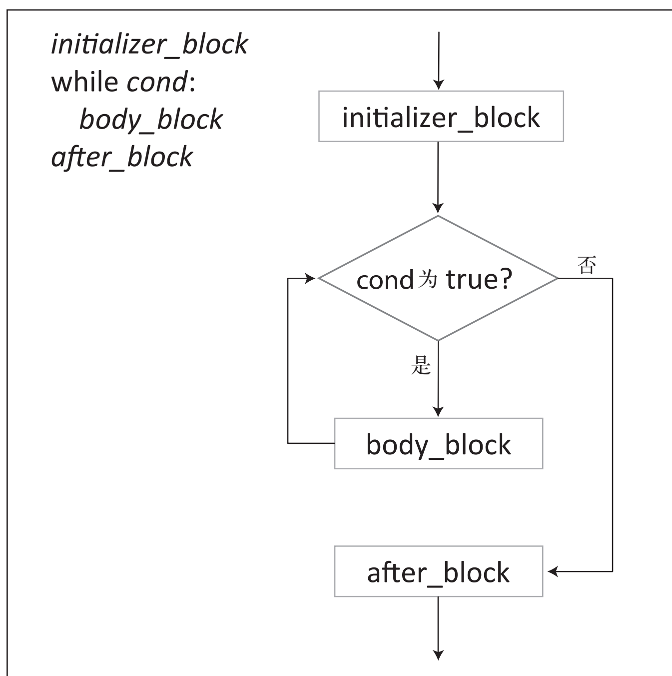


图 4-3　一个描述通用 while 循环的流程图。注意到这里没有专门列出递增语句，因为它位于 body_block 中——通常（但并非总是）位于 body_block 的末尾

提示 while 循环要灵活得多。在 while 循环前面，你可使用任何代码完成任何必要的初始化。循环条件可以是任何布尔表达式，递增语句可放在 while 循环体内的任何位置，并让它做任何你喜欢的事情。

提示 永远不会结束的循环被称为无限循环。例如，下面的循环就永不停止：

```
while True:
  print('spam')
```

提示 有些程序员喜欢将无限循环作为一种快速编写循环的方式。然而，这通常被认为是一种糟糕的风格，因为这样的循环通常复杂而难以理解。

提示 很多 Python 程序员都尽量使用 for 循环，仅在万不得已时才使用 while 循环。

提示 while 循环可包含一个 else 语句块，但这是一种不同寻常的特性，实际上很少使用，因此这里不讨论。如果你对此有兴趣，可阅读 Python 在线文档，如 http://docs.python.org/3/reference/compound_stmts.html。

4.5　比较 for 循环和 while 循环

下面来通过一些示例演示如何使用 for 循环和 while 循环解决相同的问题。另外，还将介绍一个使用 for 循环编写不出来的简单程序。

4.5.1　计算阶乘

阶乘是类似于这样的数字：$1 \times 2 \times 3 \times \cdots \times n$，指出了将 n 个物体排列时有多少种方式。例如，排列字母 ABCD 时，有 $1 \times 2 \times 3 \times 4 = 24$ 种不同的方式。下面是一种使用 for 循环计算阶乘的方式：

```
# forfact.py
n = int(input('Enter an integer >= 0: '))
fact = 1
for i in range(2, n + 1):
  fact = fact * i
print(str(n) + ' factorial is ' + str(fact))
```

下面是另一种使用 while 循环计算阶乘的方式：

```
# whilefact.py
n = int(input('Enter an integer >= 0: '))
fact = 1
i = 2
while i <= n:
  fact = fact * i
  i = i + 1
print(str(n) + ' factorial is ' + str(fact))
```

4

在用户看来，这两个程序的行为相同，但内部结构有天壤之别。与往常一样，`while` 循环版本比 `for` 循环版本要复杂些。

提示 在数学中，使用 $n!$ 来表示阶乘。例如，$4! = 1 \times 2 \times 3 \times 4 = 24$。根据定义，$0! = 1$。有趣的是，没有计算阶乘的简单公式。

提示 Python 对整数的最大取值没有限制，因此你可以使用这些程序来计算非常大的阶乘。例如，一幅扑克牌的排列方式有 52! 种：

```
Enter an integer >= 0: 52
52 factorial is 80658175170943878571
→ 6606368564037669752895054408832778
→ 24000000000000
```

4.5.2 计算用户输入的数字的总和

下面的程序让用户输入一些数字，然后打印出这些数字的总和。这是使用 for 循环的版本：

```
# forsum.py
n = int(input('How many numbers to sum? '))
total = 0
for i in range(n):
  s = input('Enter number ' +
  → str(i + 1) + ': ')
  total = total + int(s)
print('The sum is ' + str(total))
```

这是使用 while 循环的版本：

```
# whilesum.py
n = int(input('How many numbers to sum? '))
total = 0
i = 1
while i <= n:
  s = input('Enter number ' + str(i) + ': ')
  total = total + int(s)
  i = i + 1
print('The sum is ' + str(total))
```

同样，while 循环版本比 for 循环版本更复杂些。

提示　这些程序假设用户输入的是整数。如果用户输入的是浮点数，调用 int(s) 时将截短它们。当然，要支持用户输入浮点数很容易，只需将 int(s) 改为 float(s) 即可。

4.5.3 计算未知个数字的总和

下面介绍一种情形，使用前面介绍的 for 循环无法处理这种情形。假设我们要让用户输入一系列数字，以计算它们的总和，但预先并不询问用户有多少个数字。相反，用户输入所有的数字后，只需输入 'done' 来指出这一点。下面演示了如何使用 while 循环来完成这项任务。

```
# donesum.py
total = 0
s = input('Enter a number (or "done"): ')
while s != 'done':
  num = int(s)
  total = total + num
  s = input('Enter a number (or "done"): ')
print('The sum is ' + str(total))
```

这里的基本思想是，不断要求用户输入数字，直到用户输入 'done' 才罢手。这个程序预先并不知道循环体将执行多少次。

请注意下面几个细节。

- 我们必须在两个地方调用 input：循环前面及循环体内。之所以必须这样做，是因为循环条件判断输入是数字还是 'done'。
- 在循环体内，语句的排列顺序至关重要。如果循环条件为 True，就说明 s 不是 'done'，所以假定它是一个整数。因此，我们可以将它转换为整数，将其与总和相加，并让用户接着输入。
- 仅当确定输入字符串 s 不是字符串 'done' 后，我们才把它转换为整数。如果像前面那样编写下面这样的代码，则用户输入 'done' 时程序将崩溃。

```
s = int(input('Enter a number
→ (or "done"): '))
```

- 不再需要计数器变量 i。在前面计算总和的程序中，i 用于记录用户输入了多少个数字。一般而言，程序包含的变量越少，越容易理解、调试和扩展。

4

4.6 跳出循环和语句块

break 语句提供了一种便利方式，让你能够从循环体内的任何地方跳出循环。例如，下面是另一种计算未知个数字总和的方式：

```
# donesum_break.py
total = 0
while True:
  s = input('Enter a number (or "done"): ')
  if s == 'done':
    break # jump out of the loop
  num = int(s)
  total = total + num
print('The sum is ' + str(total))
```

while 循环条件为 True，这意味着将永远循环下去，直到 break 被执行。仅当 s 为 'done' 时，break 才被执行。

相比于 donesum.py，这个程序的一个优点是，没有重复 input 语句；但缺点在于终止循环的条件位于循环体中。在这样的小程序中，不难明白这一点，但在较大的程序中，break 语句可能不容易理解。另外，你可以使用任意数量的 break 语句，这让循环更难理解。

一般而言，明智的选择是，除非 break 语句让代码更简单或更清晰，否则不要使用它。

一个与 break 相关的语句是 continue 语句：在循环体中调用 continue 时，将立即跳转到循环条件，即进入循环的下一次迭代。它不像 break 那么常见，但通常也应避免使用它。

提 示 语句 break 和 continue 也可用于 for 循环中。

4.7 循环中的循环

循环中的循环也叫*嵌套循环*，在编程中相当常见。例如，下面是一个打印九九乘法表的程序：

```
# timestable.py
for row in range(1, 10):
  for col in range(1, 10):
    prod = row * col
    if prod < 10:
      print(' ', end = '')
    print(row * col, ' ', end = '')
  print()
```

请注意这个程序中的代码缩进情况：你根据缩进判断语句属于哪个代码块。最后的 `print()` 语句与第 2 条 `for` 语句对齐，这意味着它属于外部（而不是内部）`for` 循环。

注意到使用了语句 `if prod < 10` 来确保输出整齐地排列。如果没有这条语句，数字就不能对齐。

提 示 使用嵌套循环时，对循环索引变量要特别小心，以免将相同的变量用于不同循环。在大多数情况下，每个循环都需要自己的控制变量。

提 示 可根据需要在循环中嵌套任意数量的循环，但这样做将导致代码的复杂度激增。

提 示 前面说过，如果你在嵌套循环中使用 break 或 continue，break 将只跳出当前循环，而 continue 只进入当前循环的下一次迭代。

5

函　　数

函数是一大块可重用的代码。它是有名称的代码块，接受输入、提供输出并可存储在文件中供以后使用。几乎任何 Python 代码都可放在函数中。

Python 为函数提供了强大支持。例如，它提供了很多给函数传递数据的方式；它还允许在函数中包含文档字符串，让你或其他程序员能够获悉函数的工作原理。

要彻底搞懂函数，需要学习大量细节。通过实践，这些细节将很快融入你的脑海中，因此请务必尝试本章的示例。

本章内容

- 调用函数
- 定义函数
- 变量作用域
- 使用 main 函数
- 函数参数
- 模块

5.1 调用函数

在本书前面，我们一直在调用函数，下面花点时间更详细地研究一下*函数调用*。

请看内置函数 pow(x, y)，它计算 x ** y，即 x 的 y 次方：

```
>>> pow(2, 5)
32
```

其中，pow 是*函数名*，2 和 5 是传递给 pow 的实参。32 是返回值，因此我们说 pow(2, 5) 返回 32。图 5-1 概述了函数调用。当你在表达式中调用函数时，Python 将函数调用替换为其返回值，例如，表达式 pow(2, 5) + 8 与 32 + 8 等价，结果为 40。

即便函数不接受任何输入（即没有参数），也必须在函数名后添加圆括号 ()：

```
>>> dir()
['__builtins__', '__doc__', '__name__',
→ '__package__']
```

() 让 Python 执行指定的函数，如果省略 ()，输出将如下：

```
>>> dir
<built-in function dir>
```

省略了 () 时，Python 不执行函数，而告诉你 dir 指向一个函数。

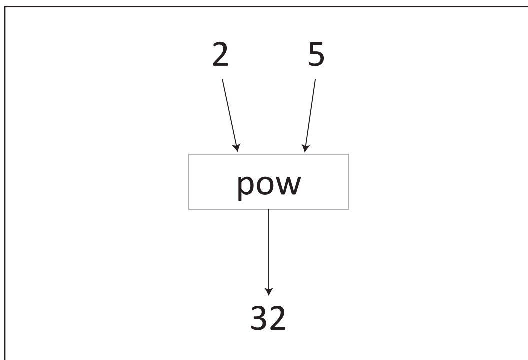


图 5-1 可将函数视为接受输入（这里为 2 和 5）并返回输出（这里为 32）的黑盒，这通常很有帮助。从调用函数 pow 的程序员的角度看，没有获悉 pow 内部结构的简单方式。我们只知道文档提供的信息，还是就是调用函数时它做什么

计算幂

pow(x, y) 与 x ** y 等价。你可能注意到了，在 Python 中，pow(0, 0)（还有 0 ** 0）的值为 1，这一点一直存在争议。有些数学家认为，pow(0, 0) 的值应该不确定或未定义；但其他一些数学家认为，将 pow(0, 0) 定义为 1 更合乎逻辑。Python 显然支持后一种观点。

5.1.1 不返回值的函数

有些函数（如 print）不返回值。请看下述代码：

```
>>> print('hello')
hello
>>> x = print('hello')
hello
>>> x
>>> print(x)
None
```

这里将特殊值 None 赋给了变量 x。None 表明“无返回值”：它既不是字符串，也不是数字，因此你不能用它来做任何有意义的计算。

5.1.2 给函数名赋值

你必须小心，以避免无意间让内置函数名指向其他函数或值。不幸的是，Python 并不会阻止你编写类似下面的代码：

```
>>> dir = 3
>>> dir
3
>>> dir()
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
  dir()
TypeError: 'int' object is not callable
```

这里让 dir 指向了数字 3，导致你再也无法访问 dir 原来指向的函数！要恢复原样，需要重启 Python。

5.2 定义函数

下面将注意力转向创建自己的函数。做为一个例子，咱们来编写一个计算圆面积的函数。你应该还记得，圆面积为 pi 乘以半径的平方。下面是一个执行这种计算的 Python 函数：

```
# area.py
import math
def area(radius):
  """ Returns the area of a circle
  with the given radius.
  For example:
  >>> area(5.5)
  95.033177771091246
  """
  return math.pi * radius ** 2
```

将这个函数保存到一个 Python 文件中（area.py 是个合适的文件名），再在 IDLE 编辑器中打开，并按 F5 运行它。如果输入无误，将出现一个提示符，但除此之外别无他物，这是因为你必须调用函数，它才会执行。要调用这个函数，只需输入函数名以及用圆括号括起的半径：

```
>>> area(1)
3.1415926535897931
>>> area(5.5)
95.033177771091246
>>> 2 * (area(3) + area(4))
157.07963267948966
```

可以像调用其他函数一样调用函数 area，差别在于函数 area 是你编写的，因此你对它做什么及如何做有控制权。

给函数命名

与变量名一样，函数名也只能包含字母、数字和下划线 _，且不能以数字打头。

一般而言，应给函数指定一个指出其用途的简单名称。函数名不应太长，也不应太短。倘若你为函数选择有益的名称，其他阅读你的代码或使用你的函数的程序员（也包括几个月后的你！）将感激不尽。

一种格式约定

Python 文档字符串通常遵循一种标准格式约定：用三引号标识文档字符串的开始和结束位置；第 1 行是函数的简要描述，对程序员很有帮助；接下来是详情和示例。

文档字符串的其他好处

与内置函数一样，你也可轻松地查看自己编写的函数的文档字符串，如下所示：

```
>>> print(area.__doc__)
Returns the area of a circle with the
➝ given radius.
For example:
  >>> area(5.5)
  95.033177771091246
```

正如你将看到的，当你在 IDLE 编辑器中调用 area 时，IDLE 将自动读取该函数的文档字符串，并将其作为工具提示显示出来。

Python 还有一个很有用的工具——doctest，可用于自动运行文档字符串中的 Python 示例代码。这是一种不错的代码测试方式，还可帮助确保文档准确地描绘了函数。本书不详细介绍 doctest，但它易于使用且非常实用，更详细的信息请参阅 http://docs.python.org/3/library/doctest.html。

函数的组成部分

来看看函数 area 的各个组成部分。第 1 行（以 def 打头的那行）被称为函数头；函数头后面所有缩进的代码被称为函数体。

函数头总是以关键字 def（definition 的缩写）打头，接下来是一个空格，然后是函数名（这里为 area）。函数命名规则与变量命名规则相同。

函数名后面是参数列表。这是一个变量列表，其中的变量指向函数的输入。函数只有一项输入——radius，但可以有任意数量的输入。如果函数没有输入，参数列表将只包含圆括号 ()。

最后，与循环和 if 语句一样，函数头也以冒号 : 结尾。

函数头后面是可选的文档字符串。文档字符串简要地描绘了函数的功能，可能包含示例和其他有益的信息。虽然文档字符串是可选的，提供它们几乎总是一个不错的主意：当你编写大量函数时，很容易忘记它们的功能和工作原理，而写得好的文档字符串可很好地提醒你。

文档字符串后面是函数体，这是一个缩进的代码块，完成了你需要完成的工作。在这个代码块中，可使用函数头中的变量。

最后，函数应使用关键字 return 返回一个值。return 语句执行时，Python 跳出函数并返回到调用这个函数的地方。

在函数 area 中，return 语句是最后一行代码，它返回使用标准公式计算得到的圆面积。注意到它在计算时使用了参数 radius，这个参数的值是在调用函数 area 时设置的。

return 语句通常是函数中最后被执行的代码（唯一的例外是，函数因本书后面将讨论的异常而意外终止）。你可将 return 语句放在函数体的任何地方，但通常放在函数末尾。

函数并非必须包含 return 语句，例如：

```
# hello.py
def say_hello_to(name):
  """ Prints a hello message.
  """
  cap_name = name.capitalize()
print('Hello ' + cap_name + ', how are you?')
```

如果函数没有包含 return 语句，Python 将认为它以下述代码行结束：

```
return None
```

特殊值 None 用于指出函数不返回值。这很常见：函数常被用于执行返回值无关紧要的任务，如在屏幕上打印输出。

术语说明

除返回值外，函数以其他任何方式所做的修改都被称为副作用（side effect）；打印到屏幕、写入文件和下载网页都属于副作用。

有一种名为函数式编程的编程风格，其特征是几乎消除了副作用。在函数式编程中，你只能通过返回值来完成修改。Python 为函数式编程提供了强有力的支持，包含在函数中定义函数以及将函数作为值传递给其他函数。使用正确的情况下，函数式编程是非常优雅而又强大的程序编写方式。

虽然本书不会详细介绍函数式编程，但尽可能避免函数带来副作用乃明智之选。

5.3 变量的作用域

函数带来的一个重要问题是作用域。变量的作用域指的是它在程序的哪些地方可访问或可见。请看下面两个函数：

```
# local.py
import math
def dist(x, y, a, b):
  s = (x - a) ** 2 + (y - b) ** 2
  return math.sqrt(s)
def rect_area(x, y, a, b):
  width = abs(x - a)
  height = abs(y - b)
  return width * height
```

5

首次赋值发生在函数内的变量被称为局部变量。函数 dist 有一个局部变量——s，而函数 rect_area 有两个局部变量——width 和 height。

函数的参数也被视为局部变量，因此 dist 总共有 5 个局部变量——x、y、a、b 和 s，而函数 rect_area 总共有 6 个局部变量。注意到变量 x、y、a 和 b 出现在这两个函数中，但它们通常指向不同的值。

局部变量只能在其所属函数中使用，这一点很重要；在函数外面，不能访问函数的局部变量。

函数结束时，其局部变量被自动删除。

全局变量

在函数外面声明的变量称为全局变量，程序中的任何函数或代码都可读取它。然而，在函数中给全局变量重复赋值时需要特别小心。请看下面的代码：

```
# global_error.py
name = 'Jack'
def say_hello():
  print('Hello ' + name + '!')
def change_name(new_name):
  name = new_name
```

第一个 name 变量是全局变量，因为它是在函数外面声明的。函数 say_hello() 读取变量 name 的值，并将其打印到屏幕上，这与你预期的一致：

```
>>> say_hello()
Hello Jack!
```

然而，调用 change_name 时结果并不符合预期：

```
>>> change_name('Piper')
>>> say_hello()
Hello Jack!
```

全局变量 name 的值并没有变，依然是 'Jack'。问题在于 Python 将函数 change_name 中的变量 name 视为局部变量，因此修改的并非全局变量 name。

要访问全局变量，必须使用关键字 global：

```
# global_correct.py
name = 'Jack'
def say_hello():
  print('Hello ' + name + '!')
def change_name(new_name):
  global name
  name = new_name
```

这样结果将截然不同，两个函数的行为都符合预期：

```
>>> say_hello()
Hello Jack!
>>> change_name('Piper')
>>> say_hello()
Hello Piper!
```

> **其他语言中的 main 函数**
>
> 使用 main 函数的做法非常普遍，有些编程语言（最著名的是 C、C++ 和 Java）还要求必须使用 main 函数。但在 Python 中，main 只是一种约定，完全是可选的。

5.4 使用 main 函数

在你编写的任何 Python 程序中，通常都至少应该使用一个函数：main()。根据约定，main() 函数被认为是程序的起点。例如，编写前一章的密码检查程序时，可使用 main 函数：

```
# password2.py
def main():
  pwd = input('What is the password? ')
  if pwd == 'apple':
    print('Logging on ...')
  else:
    print('Incorrect password.')
  print('All done!')
```

注意到所有代码都位于函数头 main 后面，并缩进了。

在 IDLE 中运行 password2.py 时，什么都不会发生，而只是出现一个提示符。你必须输入 main() 来执行其代码。

使用 main 函数的优点是，可更轻松地反复运行程序，还可传递输入值。

5.5 函数的参数

参数用于向函数传递数据，Python 支持多种参数。

5.5.1 按引用传递

向函数传递参数时，Python 采用按引用传递的方式。这意味着当你传递参数时，函数将使用新变量名来引用原始值。例如，请看下面这个简单的程序：

```
# reference.py
def add(a, b):
  return a + b
```

启动 IDLE 的交互式命令行，并这样做：

```
>>> x, y = 3, 4
>>> add(x, y)
7
```

在第 1 行设置 x 和 y 后，内存类似于图 5-2 所示。当调用 add(x,y) 时，Python 创建两个新变量——a 和 b，它们分别指向 x 和 y 的值，如图 5-3 所示。按排列顺序进行赋值，因此 a 指向 x，因为 x 是第一个实参。注意到没有复制实参的值，而只是给它们指定新名称，而函数将使用这些新名称来引用它们。

将 a 和 b 相加后，函数返回，而 a 和 b 被自动删除。在整个函数调用过程中，x 和 y 未受影响。

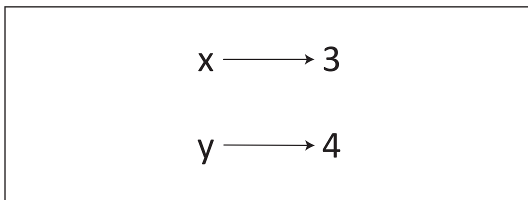


图 5-2　将 x 和 y 分别设置为 3 和 4 后的内存状态

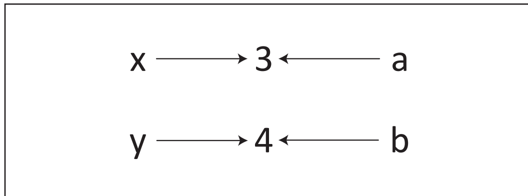


图 5-3　刚调用 add(x,y) 后的内存状态：a 和 b 分别指向 x 和 y 指向的值

按值传递

有些编程语言（如 C++）可按值传递参数。按值传递参数时，将创建其拷贝，并将该拷贝传递给函数。如果传递的值很大，复制可能消耗大量时间和内存。Python 不支持按值传递。

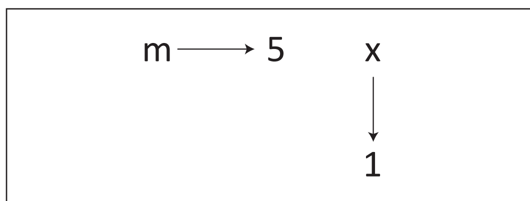


图 5-4　在函数调用 set1(m) 中将 1 赋给 x 后，m 并没变，依然指向原来的值 5，但局部变量 x 确实被设置成了 1

5.5.2　一个重要示例

按引用传递简单而高效，但有些事情它做不了。例如，请看下面这个名不副实的函数：

```
# reference.py
def set1(x):
  x = 1
```

函数 set1 想将传入的变量的值设置为 1，但如果你尝试运行它，结果并不符合预期：

```
>>> m = 5
>>> set1(m)
>>> m
5
```

m 的值根本没变，太令人意外了。这都是按引用传递导致的。为帮助理解，将这个示例分解成下面几步：

1. 将 5 赋给 m。

2. 调用 set1(m)：将 m 的值赋给 x，这样 m 和 x 都指向 5。

3. 将 1 赋给 x，结果如图 5-4 所示。

4. 函数 set1 结束后，x 被删除。

在函数 set1 中，根本不能访问变量 m，因此无法修改它指向的值。

5.5.3　默认值

给参数指定默认值通常很有帮助。例如，在下面的函数中，给参数 greeting 指定了默认值 'Hello'：

```
# greetings.py
def greet(name, greeting = 'Hello'):
  print(greeting, name + '!')
```

这让你能够以两种不同的方式调用函数 greet：

```
>>> greet('Bob')
Hello Bob!
>>> greet('Bob', 'Good morning')
Good morning Bob!
```

提示　默认参数很方便，在 Python 中相当常见。

提示　函数可根据需要使用任意数量的默认参数，但带默认值的参数不能位于没有默认值的参数前面。

提示　有关默认参数的一个要点是，只在第一次调用函数时给默认参数赋值。在复杂的程序中，这可能成为微妙 bug 的根源，因此牢记这一点很有帮助。

5.5.4　关键字参数

在 Python 中，另一种很有用的参数指定方式是使用关键字。例如：

```
# shopping.py
def shop(where = 'store',
         what = 'pasta',
         howmuch = '10 pounds'):
         print('I want you to go to the',
         → where)
         print('and buy', howmuch, 'of',
         → what + '.')
```

调用使用关键字参数的函数时，以 param = value 的方式传递数据。例如：

```
>>> shop()
I want you to go to the store
and buy 10 pounds of pasta.
>>> shop(what = 'towels')
I want you to go to the store
and buy 10 pounds of towels.
>>> shop(howmuch = 'a ton', what = 'towels')
I want you to go to the store
and buy a ton of towels.
>>> shop(howmuch = 'a ton', what = 'towels',
→ where = 'bakery')
I want you to go to the bakery
and buy a ton of towels
```

关键字参数有两大好处。首先，它们清晰地指出了参数值，有助于提高程序的可读性；其次，关键字参数的顺序无关紧要。对于包含大量参数的函数来说，这两点都很有帮助，因为很难记住这些函数的参数的顺序和含义。

5.6 模块

模块是一系列相关的函数和变量。

5.6.1 创建 Python 模块

要创建模块，可创建一个 .py 文件，在其中包含用于完成任务的函数。例如，下面是一个简单的模块，用于在屏幕上打印形状：

```
# shapes.py
"""A collection of functions
  for printing basic shapes.
"""
CHAR = '*'
def rectangle(height, width):
  """ Prints a rectangle. """
  for row in range(height):
    for col in range(width):
      print(CHAR, end = '')
    print()
def square(side):
  """ Prints a square. """
  rectangle(side, side)
def triangle(height):
  """ Prints a right triangle. """
  for row in range(height):
    for col in range(1, row + 2):
      print(CHAR, end = '')
    print()
```

模块与常规 Python 程序之间唯一的差别是用途不同：模块是一个由函数组成的工具箱，用于编写其他程序。因此，模块通常没有 `main()` 函数。

要使用模块，只需导入它即可。例如：

```
>>> import shapes
>>> dir(shapes)
['CHAR', '__builtins__', '__doc__',
→ '__file__', '__name__', '__package__',
→ 'rectangle', 'square', 'triangle']
>>> print(shapes.__doc__)
A collection of functions
for printing basic shapes.
>>> shapes.CHAR
'*'
>>> shapes.square(5)
*****
*****
*****
*****
*****
>>> shapes.triangle(3)
*
**
***
```

你还可导入模块的所有内容：

```
>>> from shapes import *
>>> rectangle(3, 8)
********
********
********
```

5.6.2 名称空间

对于模块，很有用的一点是它们形成名称空间。名称空间基本上就是一组独特的变量名和函数名。要让模块中的名称在模块外面可见，你必须使用 import 语句。

为了了解这为何很重要，我们假设 Jack 和 Sophie 合作开发一个大型编程项目。Jack 住在西海岸，而 Sophie 住在东海岸。他们达成了一致，Jack 将其所有代码都放在模块 jack.py 中，Sophie 将其所有代码都放在模块 sophie.py 中。他们各自独立地工作，最终都编写了函数 save_file(fname)，但这两个函数只是函数头相同，功能截然不同。这两个函数位于不同的模块，即便同名也没有关系，因为它们位于不同的名称空间。Jack 编写的函数的全名为 jack.save_file(fname)，而 Sophie 编写的函数的全名为 sophie. save_file(fname)。

模块可避免名称冲突，让程序员能够独立地进行开发。即便你不与其他程序员合作，名称冲突也是个恼人的问题，在程序很大很复杂时尤其如此。

当然，即便使用了模块，像下面这样做依然会导致名称冲突：

```
>>> from jack import *
>>> from sophie import *
```

这种 import 语句将每个模块的所有内容都加入到当前名称空间，将覆盖其他同名的内容。因此，在较大的程序中，明智的做法是不使用 from ... import * 语句。

Python 之禅

要查看一个有趣的 Python 复活节彩蛋，可尝试从交互式命令行导入模块 this：

```
>>> import this
```

6 字　符　串

在 Python 中，字符串是除数字外最重要的数据类型。字符串无处不在：将字符串打印到屏幕；从用户那里读取字符串；正如你将在第 8 章看到的，文件通常被视为大型字符串。可将万维网视为一个网页集合，而这些网页大部分是由文本组成的。

字符串是一种聚合数据结构，这让我们有机会初探索引和切片——用于从字符串中提取子串的方法。

本章还将简要地介绍 Python 正则表达式库，这是一种设计来用于处理字符串的微型语言，但动力强劲。

本章内容

- 字符串索引
- 字符
- 字符串切片
- 标准字符串函数
- 正则表达式

6.1 字符串索引

第 2 章介绍过字符串，如果你要复习字符串基本知识，可回过头去阅读相关内容。

处理字符串时，经常需要访问其中的各个字符。例如，假设你知道 s 是一个字符串，并想打印其中的各个字符，则可使用字符串索引：

```
>>> s = 'apple'
>>> s[0]
'a'
>>> s[1]
'p'
>>> s[2]
'p'
>>> s[3]
'l'
>>> s[4]
'e'
```

Python 使用方括号来标识字符串索引：方括号内的数字指出了要获取哪个字符，如图 6-1 所示。在 Python 中，最小的字符串索引总是 0，而最大的索引总是比字符串长度小 1。

因此，如果 s 指向一个长度为 n 的字符串，则 s[0] 为第一个字符，s[1] 为第二个字符，s[2] 为第三个字符，依此类推。s[n-1] 为最后一个字符。

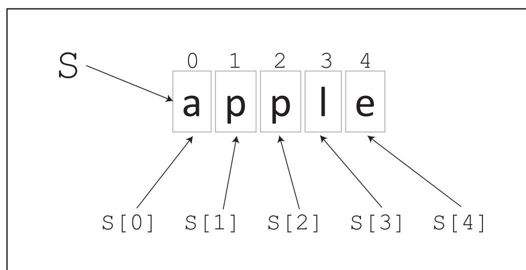


图 6-1 字符串 'apple' 的索引值，方括号索引表示法用于访问字符串的各个字符

为何从零开始

Python 索引从零开始，初学者常常觉得怪怪的。要习惯这一点，确实需要时间。它还可能导致困扰众多程序员的“差一错误”。这样想对理解它很有帮助：索引值用于测量与字符串第一个字符相隔的距离，就像一把尺子（其刻度也是从零开始）。这让有些索引计算更简单，也与函数 %（求余）一致。% 经常用于索引计算，自然也可能返回 0。

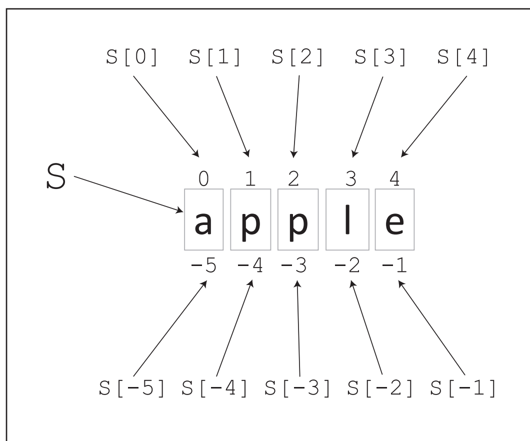


图 6-2 Python 字符串有正索引和负索引；实际上，程序员通常使用最方便的索引

如果索引超出了字符串末尾，将导致“超出范围”错误：

```
>>> s[5]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
  s[5]
IndexError: string index out of range
```

6.1.1 负数索引

假设你要访问 s 的最后一个字符，而不是第一个字符。为此，可使用难看的表达式 `s[len(s)-1]`，这当然可行，但相当复杂。所幸的是，在访问字符串末尾附近的字符方面，Python 提供了一种更便利的方式：*负数索引*。其理念是，沿从右向左的方向，用负数表示字符串中字符的索引：

```
>>> s = 'apple'
>>> s[-1]
'e'
>>> s[-2]
'l'
>>> s[-3]
'p'
>>> s[-4]
'p'
>>> s[-5]
'a'
```

因此，字符串的最后一个字符为 `s[-1]`。图 6-2 说明了负数索引的工作原理。

6.1.2 使用 for 循环访问字符

如果需要依次访问字符串的每个字符，for 循环可助一臂之力。例如，下面的程序计算给定字符串的字符编码总和：

```
# codesum.py
def codesum1(s):
  """ Returns the sums of the
  character codes of s.
  """
  total = 0
  for c in s:
    total = total + ord(c)
  return total
```

下面是一个调用示例：

```
>>> codesum1('Hi there!')
778
```

像这样使用 for 循环时，在循环的每次迭代开头，都会将循环变量 c 设置为 s 中的下一个字符。使用索引访问 s 中字符的工作由 for 循环自动处理。

下面是另一种实现，它使用了常规字符串索引，请将其与 codesum1 进行比较：

```
def codesum2(s):
  """ Returns the sums of the
  character codes of s.
  """
  total = 0
  for i in range(len(s)):
    total = total + ord(s[i])
  return total
```

这种实现的结果与 codesum1 相同，但更复杂、可读性更差。

Unicode 的崛起

在 20 世纪 60 年代到 80 年代，最流行的字符编码方案是 ASCII（美国信息交换标准编码）。ASCII 比 Unicode 简单得多，但存在一个致命缺陷，那就是只能表示 256 个字符。对英语、法语和其他几种类似的语言来说，这足够了，但要表示其他语言中众多的符号和字符，这远远不够。例如，可能出现在文本文档中的表意中文字就有数千个。

Unicode 提供了一个大得多的字符编码集。出于方便考虑，Unicode 的前 256 个字母为 ASCII 码，因此如果你只处理英文字符，几乎不用考虑 Unicode 的细节。有关 Unicode 的更详细信息，请参阅其主页 www.unicode.org。

6.2 字符

字符串由字符组成，而字符本身实际上是一个非常复杂的问题。正如第 2 章指出的，所有字符都有对应的字符编码，你可以使用函数 ord 来获悉：

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('c')
99
```

给定字符编码，可使用函数 chr 来获悉对应的字符：

```
>>> chr(97)
'a'
>>> chr(98)
'b'
>>> chr(99)
'c'
```

字符编码是根据 Unicode 分配的，而 Unicode 是一个庞杂的编码标准，涵盖了全球各种语言中的符号和字符。

转义字符

并非所有字符都有可视的标准符号。例如，换行字符、回车字符和制表符都是不可见的，虽然它们带来的效果可见。这些字符属于空白字符——在印刷页面上显示为空白。

为处理空白字符以及其他不可打印的字符，Python 使用一种特殊表示法——转义序列（也叫转义字符）。表 6-1 列出了最常用的转义字符。

要在字符串中包含反斜杠、单引号和双引号，通常需要使用对应的转义字符。例如：

```
>>> print('\' and \" are quotes')
' and " are quotes
>>> print('\\ must be written \\\\')
\ must be written \\
```

在 Python 中，表示换行的标准方式是使用字符 \n：

```
>>> print('one\ntwo\nthree')
one
two
three
```

转义字符是单个字符，明白这一点很重要。为让 Python 知道下一个字符是特殊字符，必须使用 \，但在计算字符串的长度时，并不将 \ 视为额外的字符。例如：

```
>>> len('\\')
1
>>> len('a\nb\nc')
5
```

表 6-1　一些常见的转义字符

字　　符	含　　义
\\	反斜杠
\'	单引号
\"	双引号
\n	换行符
\r	回车
\t	水平制表符

行尾

在表示文本行末尾方面，不同操作系统遵循的标准也不同。例如，Windows 使用 \r\n 表示行尾，OS X 和 Linux 使用 \n，而 OS X 之前的 Mac 操作系统使用 \r。

大多数优秀编辑器都至少能够识别 \r\n 和 \n。你会偶尔遇到不能识别行尾标记的程序（如 Windows“记事本”），因此可能出现这样的情形：所有内容都出现在一行、增加了换行或每行末尾都有 ^M 字符。避免这种问题的最简单方式是，使用能正确处理行尾标记的文本编辑器。

6.3 字符串切片

在 Python 中，可使用切片从字符串中提取子串。要对字符串执行切片操作，可指定两个索引：要提取的第一个字符的索引；要提取的最后一个字符的索引加 1。例如：

```
>>> food = 'apple pie'
>>> food[0:5]
'apple'
>>> food[6:9]
'pie'
```

用于切片的索引与用于访问各个字符的索引相同：第一个索引总是为零，而最后一个索引总是比字符串长度小 1。一般而言，s[begin:end] 返回从索引 begin 到 end-1 的子串。

请注意，如果 s 是一个字符串，则要访问索引 i 对应的字符，可使用 s[i] 或 s[i:i+1]。

6.3.1　获取切片的捷径

如果你省略切片的起始索引，Python 将假定它为零；如果你省略切片的终止索引，Python 将假设你要提取到字符串末尾。例如：

```
>>> food = 'apple pie'
>>> food[:5]
'apple'
>>> food[6:]
'pie'
>>> food[:]
'apple pie'
```

下面是一个很有用的切片示例，这个函数返回文件名中的扩展名：

```
# extension.py
def get_ext(fname):
  """ Returns the extension of file
  fname.
  """
  dot = fname.rfind('.')
  if dot == -1: # fname 中没有
    return ''
  else:
    return fname[dot + 1:]
```

下面演示了函数 get_ext 的作用：

```
>>> get_ext('hello.text')
'text'
>>> get_ext('pizza.py')
'py'
>>> get_ext('pizza.old.py')
'py'
>>> get_ext('pizza')
''
```

函数 get_ext 的工作原理如下：确定最右边的 '.' 的索引（因此使用 rfind 从右往左查找）；如果 fname 不包含 '.'，则返回一个空字符串，否则返回 '.' 后面的所有字符。

6.3.2 使用负数索引的切片

执行切片操作时，也可使用负数索引，虽然这更难懂。例如：

```
>>> food = 'apple pie'
>>> food[-9:-4]
'apple'
>>> food[:-4]
'apple'
>>> food[-3:0]
''
>>> food[-3:]
'pie'
```


使用负数索引时，这样做通常会有所帮助：将字符串写到纸上，再标出每个字符的正索引和负索引，就像图 6-2 那样。虽然这样做确实需要多用一两分钟时间，但可以很好地避免常见的索引错误。

6.4 标准字符串函数

Python 字符串自带了大量很有用的函数，要查看所有这些函数，可调用 `dir` 并将参数指定为任何字符串（如 `dir('')`）。虽然没有必要准确地记住所有这些函数的功能，但最好大致了解它们的功能，这样才能根据需要使用它们。因此，本节将分门别类地介绍字符串自带的所有函数。

这里无意提供完整的参考手册：对一些不太常用的参数，将省略不表，也不详细介绍每个函数的所有细节。要更详细地了解这些函数，可参阅其文档字符串或 Python 在线文档（http://docs.python.org/3/）。

6.4.1 测试函数

首先介绍检测字符串是否为特定格式的函数，它们组成了一个最大的字符串函数组。表 6-2 所示的测试函数都返回 `True` 或 `False`。测试函数有时也叫布尔函数或谓词。

表 6-2 字符串测试函数

函数名	什么情况下返回 True
`s.endswith(t)`	s 以字符串 t 结尾
`s.startswith(t)`	s 以字符串 t 打头
`s.isalnum()`	s 只包含字母或数字
`s.isalpha()`	s 只包含字母
`s.isdecimal()`	s 只包含表示十进制数字的字符
`s.isdigit()`	s 只包含数字字符
`s.isidentifier()`	s 是合法的标识符
`s.islower()`	s 只包含小写字母
`s.isnumeric()`	s 只包含数字
`s.isprintable()`	s 只包含可打印字符
`s.isspace()`	s 只包含空白字符
`s.istitle()`	s 是个大小写符合头衔要求（title-case）的字符串
`s.isupper()`	s 只包含大写字母
`t in s`	s 包含字符串 t

表 6-3　字符串搜索函数

函　数　名	返　回　值
s.find(t)	如果没有找到子串 t，则返回 -1；否则返回 t 在 s 中的起始位置
s.rfind(t)	与 find 相同，但从右往左搜索
s.index(t)	与 find 相同，但如果在 s 中找不到 t，则引发 ValueError 异常
s.rindex(t)	与 index 相同，但从右往左搜索

6.4.2　搜索函数

如表 6-3 所示，在字符串中查找子串的方式有多种。函数 index 和 find 之间的差别在于没有找到指定子串时的情形。例如：

```
>>> s = 'cheese'
>>> s.index('eee')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
  s.index('eee')
ValueError: substring not found
>>> s.find('eee')
-1
```

函数 index 引发异常 ValueError，第 9 章将更详细地介绍异常。如果没有找到指定的子串，函数 find 将返回 -1。

字符串搜索函数通常从左往右（从开头往末尾）搜索，但以 r 打头的函数从右往左搜索。例如：

```
>>> s = 'cheese'
>>> s.find('e')
2
>>> s.rfind('e')
5
```

一般而言，函数 find 和 index 返回传入字符串第一次出现时的起始位置索引，而 rfind 和 rindex 返回传入字符串最后一次出现时的起始位置索引。

6.4.3　改变大小写的函数

Python 提供了各种修改字母大小写的函数，如表 6-4 所示。别忘了，Python 绝不会修改字符串：在所有这些函数中，Python 都创建并返回一个新字符串。我们谈论字符串时，常常让人觉得字符串被修改，但这只是为了方便措词，并不意味着字符串真的被修改。

表 6-4　改变大小写的函数

函　数　名	返回的字符串
s.capitalize()	将 s[0] 改为大写
s.lower()	让 s 的所有字母都小写
s.upper()	让 s 的所有字母都大写
s.swapcase()	将小写字母改为大写，并将大写字母改为小写
s.title()	让 s 的大小写符合头衔的要求

6.4.4　设置格式的函数

表 6-5 列出了设置字符串格式的函数，它们让你能够美化字符串，以便向用户显示或打印到文件。

表 6-5　设置字符串格式的函数

函　数　名	返回的字符串
s.center(n, ch)	包含 n 个字符的字符串，其中 s 位于中央，两边用字符 ch 填充
s.ljust(n, ch)	包含 n 个字符的字符串，其中 s 位于左边，右边用字符 ch 填充
s.rjust(n, ch)	包含 n 个字符的字符串，其中 s 位于右边，左边用字符 ch 填充
s.format(vars)	见正文

字符串函数 format 功能非常强大，包含用于设置字符串格式的微型语言。要使用函数 format，你需要给它提供变量或值，如下例所示：

```
>>> '{0} likes {1}'.format('Jack', 'ice cream')
'Jack likes ice cream'
```

字符串中的 {0} 和 {1} 引用 format 的参数：它们将被替换为相应字符串或变量的值。你还可以使用关键字参数的名称：

```
>>> '{who} {pet} has fleas'.format (pet =
'dog',
→ who = 'my')
'my dog has fleas'
```

这些示例演示了 format 的最基本用法，还有很多其他的选项，可用于指定字符串间距、将数字转换为字符串等。所有细节都可在 Python 在线文档中找到，网址为 http://docs.python.org/3/library/string.html#format-string-syntax。

表 6-6　字符串剥除函数

函　数　名	返回的字符串
s.strip(ch)	从 s 开头和末尾删除所有包含在字符串 ch 中的字符
s.lstrip(ch)	从 s 开头（左端）删除所有包含在字符串 ch 中的字符
s.rstrip(ch)	从 s 末尾（右端）删除所有包含在字符串 ch 中的字符

表 6-7　字符串拆分函数

函　数　名	返回的字符串
s.partition(t)	将 s 拆分为三个字符串（head、t 和 tail），其中 head 为 t 前面的子串，而 tail 为 t 后面的子串
s.rpartition(t)	与 partition 相同，但从 s 的右端开始搜索 t
s.split(t)	以 t 为分隔符，将 s 划分成一系列子串，并返回一个由这些子串组成的列表
s.rsplit(t)	与 split 相同，但从 s 的右端开始搜索 t
s.splitlines()	返回一个由 s 中的各行组成的列表

6.4.5　剥除函数

剥除函数用于删除字符串开头或末尾多余的字符，如表 6-6 所示。默认情况下，将剥除空白字符；如果指定了一个字符串参数，将剥除该字符串中的字符，如下所示：

```
>>> name = ' Gill Bates '
>>> name.lstrip()
'Gill Bates '
>>> name.rstrip()
' Gill Bates'
>>> name.strip()
'Gill Bates'
>>> title = '_-_- Happy Days!! _-_-'
>>> title.strip()
'_-_- Happy Days!! _-_-'
>>> title.strip('_-')
' Happy Days!! '
>>> title.strip('_ -')
'Happy Days!!'
```

6.4.6　拆分函数

拆分函数将字符串拆分成多个子串，如表 6-7 所示。函数 partition 和 rpartition 将字符串拆分为 3 部分：

```
>>> url = 'www.google.com'
>>> url.partition('.')
('www', '.', 'google.com')
>>> url.rpartition('.')
('www.google', '.', 'com')
```

这两个函数总是返回一个这样的值：它由 3 个字符串组成，形式为 (head, sep, tail)。这种返回值为元组，将在第 7 章更详细地介绍。

函数 split 以指定字符串为分隔符，将字符串划分为一系列子串，如下所示：

```
>>> url = 'www.google.com'
>>> url.split('.')
['www', 'google', 'com']
>>> story = 'A long time ago, a princess ate
→ an apple.'
>>> story.split()
['A', 'long', 'time', 'ago,', 'a', 'princess',
→ 'ate', 'an', 'apple.']
```

函数 split 总是返回一个字符串列表；Python 列表总是分别以 [和] 打头和结尾，并用逗号分隔元素。正如你将在第 7 章看到的，列表和元组很像，主要差别在于列表是可以修改的，而元组是常量。

6.4.7 替换函数

Python 字符串自带了两个替换函数，如表 6-8 所示。注意到使用替换函数可轻松地删除字符串中的子串：

```
>>> s = 'up, up and away'
>>> s.replace('up', 'down')
'down, down and away'
>>> s.replace('up', '')
', and away'
```

表 6-8 字符串替换函数

函　数　名	返回的字符串
s.replace(old, new)	将 s 中的每个 old 替换为 new
s.expandtabs(n)	将 s 中的每个制表符替换为 n 个空格

6.4.8 其他函数

最后，表 6-9 列出了其他字符串函数。

需要将一组字符转换为另一组字符时，函数 translate 和 maketrans 很实用。例如，下面是一种将字符串转换为“脑残体”（leet-speak）的方式：

```
>>> leet_table = ''.maketrans ('EIOBT',
➝ '31087')
>>> 'BE COOL. SPEAK LEET!'.translate
➝ (leet_table)
'83 COOL. SP3AK L337!'
```

在线文档（http://docs.python.org/3/library/stdtypes.html#str.maketrans）还解释了如何替换多个字符。函数 zfill 用于设置数值字符串的格式：

```
>>> '23'.zfill(4)
'0023'
>>> '-85'.zfill(5)
'-0085'
```

然而，这个函数不是很灵活，因此大多数程序员宁愿使用其他方式来设置字符串的格式。

函数 join 很有用，它将一系列字符串拼接起来，其中包含分隔字符串，如下所示：

```
>>> ' '.join(['once', 'upon', 'a', 'time'])
'once upon a time'
>>> '-'.join(['once', 'upon', 'a', 'time'])
'once-upon-a-time'
>>> ''.join(['once', 'upon', 'a', 'time'])
'onceuponatime'
```

表 6-9 其他字符串函数

函　数　名	返回的值
s.count(t)	t 在 s 中出现的次数
s.encode()	设置 s 的编码，更详细的信息请参阅在线文档（http://docs.python.org/3/library/stdtypes.html#str.encode）
s.join(seq)	使用 s 将 seq 中的字符串连接成一个字符串
s.maketrans(old,new)	创建一个转换表，用于将 old 中的字符改为 new 中相应的字符；请注意，s 可以是任何字符串，它不影响返回的转换表
s.translate(table)	使用指定转换表（使用 maketrans 创建的）对 s 中的字符进行替换
s.zfill(width)	在 s 左边添加足够多的 0，让字符串的长度为 width

6.5 正则表达式

虽然 Python 字符串提供了众多实用的函数，但实际处理字符串时，常常需要使用更强大的工具。

有鉴于此，程序员开发了一种用于复杂字符串处理的微型语言——正则表达式。

实际上，正则表达式就是一种简练描绘一组字符串的方式，可用于高效地执行常见的字符串处理任务，如匹配、分拆和替换。本节介绍正则表达式的基本概念以及一些常用的运算符（如表 6-10 所示）。

表 6-10　一些正则表达式运算符

运　算　符	描述的字符串
xy?	x、xy
x|y	x、y
x*	''、x、xx、xxx、xxxx 等
x+	x、xx、xxx、xxxx 等

6.5.1 简单的正则表达式

来看字符串 'cat'，它表示单个字符串，这个字符串由字母 c、a 和 t 组成。接下来看看整个表达式 'cats?'，其中的 ? 并非英语中的问号，而是一个正则表达式运算符，意思是它左边的那个字符是可选的。因此，正则表达式 'cats?' 描述了两个字符串：'cat' 和 'cats'。

另一个正则表达式运算符是 |，其含义是“或者”。例如，正则表达式 'a|b|c' 描述了三个字符串：'a'、'b' 和 'c'。正则表达式 'a*' 描述了无穷个字符串：''、'a'、'aa'、'aaa'、'aaaa'、'aaaaa' 等。换句话说，'a*' 描述的是由零或更多个 'a' 组成的字符串。正则表达式 'a+' 与 'a*' 相同，但排除了空字符串 ''。

最后，在正则表达式中，可使用圆括号来指出要将运算符用于那个子串。例如，正则表达式 '(ha)+!' 描述了如下字符串：'ha!'、'haha!'、'hahaha!' 等；而 'ha+!' 描述了一组截然不同的字符串：'ha!'、'haa!'、'haaa!' 等。

可随意混合使用这些（以及其他）正则表达式运算符。事实证明，这是一种非常实用的方式，可描述众多常见的字符串类型，如电话号码和电子邮件地址。

6.5.2 使用正则表达式匹配字符串

匹配字符串是正则表达式的一种常见用途。例如，假设你要编写一个程序，要求用户必须输入 done 或 quit 来结束程序。为帮助识别这些字符串，可编写一个类似于下面的函数：

```
# allover.py
def is_done1(s):
  return s == 'done' or s == 'quit'
```

使用正则表达式时，行为完全与此相同的函数可能类似于下面这样：

```
# allover.py
import re # 使用正则表达式
def is_done2(s):
  return re.match('done|quit', s)!= None
```

在这个新版本中，第 1 行导入 Python 的标准正则表达式库。为匹配正则表达式，使用了函数 re.match(regex, s)，它在 regex 与 s 不匹配时返回 None，否则返回一个特殊的正则表达式匹配对象。在这个示例中，我们不关心匹配对象的细节，因此只检查返回值是否为 None。

在这样的简单示例中，正则表达式版本并不比第一个版本短很多，也不比第一个版本好很多；实际上，is_done1 可能更好！然而，随着程序越来越大、越来越复杂，正则表达式的优势就会逐渐显现出来。例如，假设我们决定再添加几个用于结束程序的字符串。对于正则表达式版本中，只需重写正则表达式字符串，如改为 'done|quit|over|finished|end|stop'；然而，要对第一个版本做同样的修改，需要为每个新增的字符串添加 or s==，这将导致代码行很长，可读性极差。

再举一个更复杂的例子。假设要识别逗人的字符串：开头为一个或多个 'ha'，末尾为一个或多个 '!'，如 'haha!'、'ha!!!!!' 和 'hahaha!!'。使用正则表达式匹配这些字符串很容易：

```
# funny.py
import re
def is_funny(s):
  return re.match('(ha)+!+', s)!= None
```

注意到 is_funny 与 is_done2 的唯一差别是，调用函数 match 时指定的正则表达式不同。如果你尝试在不使用正则表达式的情况下编写这个函数，将很快明白 '(ha)+!+' 为我们做了多少工作。

6.5.3 其他正则表达式

这里只涉及正则表达式的一些皮毛：Python 库 `re` 规模庞大，其中有大量正则表达式函数可用于执行字符串处理任务，如匹配、分拆和替换；还有提高常用正则表达式处理速度的技巧，以及众多匹配常用字符的捷径。模块 `re` 的文档（http://docs.python.org/3/library/re.html）提供了其他一些示例。

7

数据结构

本章介绍重要的数据结构概念：值集合及常用的函数。Python 秉承方便程序员的理念，提供了几个功能强大而高效的数据结构：元组、列表、字典和集合，程序员可根据需要组合使用它们，以创建更复杂的数据结构。

前一章讨论了字符串，可将其视为只存储字符序列的数据结构。本章介绍的数据结构并非只能存储字符，而是能存储几乎任何数据。

在 Python 中，两个主力数据结构是列表和字典。列表按顺序存储数据，而字典像小型数据库，使用键高效地存储和检索数据。

本章内容

- ❑ type 命令
- ❑ 序列
- ❑ 元组
- ❑ 列表
- ❑ 列表函数
- ❑ 列表排序
- ❑ 列表解析
- ❑ 字典
- ❑ 集合（set）

7.1 type 命令

在有些情况下，需要检查值或变量的数据类型。这很容易，只需使用内置命令 type：

```
>>> type(5)
<class 'int'>
>>> type(5.0)
<class 'float'>
>>> type('5')
<class 'str'>
>>> type(None)
<class 'NoneType'>
>>> type(print)
<class 'builtin_function_or_method'>
```

注意到在 type 命令的输出中，使用了术语类（class）。粗略地说，类和类型是同义词。

type 命令对调试很有帮助。例如，在 Python 中，经常需要使用数据集合，但不知道其元素甚至容器本身的数据类型，通过使用 type，可获悉 Python 对象的准确类型。

7.2 序列

在 Python 中，序列是一组按顺序排列的值。Python 有 3 种内置的序列类型：字符串、元组和列表。

序列的优点之一是，像前一章介绍的字符串一样支持索引和切片。因此，所有序列都具备如下特征。

- 第一个正索引为零，指向左端。
- 第一个负索引为 –1，指向右端。
- 可使用切片表示法来复制子序列，例如，`seq[begin:end]` 复制 `seq` 的如下元素：从索引 `begin` 指向的元素到索引 `end - 1` 指向的元素。
- 可使用 + 和 * 进行拼接（即合并）。要进行拼接，序列的类型必须相同，即不能拼接元组和列表。
- 可使用函数 `len` 计算其长度，例如，`len(s)` 返回序列 `s` 包含的元素数。
- 表达式 `x in s` 检查序列 `s` 是否包含元素 `x`。换句话说，如果 `x` 位于 `s` 中，则 `x in s` 返回 `True`，否则返回 `False`。

实际上，字符串和列表是最常用的序列；元组有其独特用途，但不那么常见。

顺序很重要

我们说序列是按顺序排列的，指的是序列中元素的排列顺序很重要。字符串是按顺序排列的，因为 'abc' 不同于 'acb'。你将在本章后面看到，字典和集合并不是按顺序排列的：它们只关心自己是否包含特定元素，而不保证元素的相对顺序。

序列可以有多长

从理论上说，序列的长度不受限制，可根据需要包含任意数量的元素；但实际上，序列长度受制于运行程序的计算机的可用内存量。

7.3 元组

元组是一种不可变序列，包含零个或更多个值。它可以包含任何 Python 值，甚至可以包含其他元组。例如：

```
>>> items = (-6, 'cat', (1, 2))
>>> items
(-6, 'cat', (1, 2))
>>> len(items)
3
>>> items[-1]
(1, 2)
>>> items[-1][0]
1
```

正如你看到的，元组用圆括号括起，其中的元素用逗号分隔。空元组用 () 表示，但只包含一个元素的元组（单元素元组）采用不同寻常的表示法 (x,)，如下所示：

```
>>> type(())
<class 'tuple'>
>>> type((5,))
<class 'tuple'>
>>> type((5))
<class 'int'>
```

如果省略单元素元组末尾的逗号，就不是创建元组，而是用圆括号将表达式括起。

末尾的逗号

在单元素元组中，末尾的逗号必不可少;而在包含更多元素的元组（和列表）中，可以在末尾添加逗号，但并非必须这样做。例如，(1,2,3,) 与 (1,2,3) 等价。有些程序员喜欢总是在末尾添加逗号，以免无意间遗漏单元素元组末尾的逗号。

7.3.1 元组是不可变的

前面说过，元组是不可变的，这意味着创建元组后就不能修改它。这种特征并不独特，字符串、整数和浮点数都是不可变的。如果要修改元组，就必须创建一个体现更改的新元组。例如，下面的代码演示了如何删除元组的第一个元素：

```
>>> lucky = (6, 7, 21, 77)
>>> lucky
(6, 7, 21, 77)
>>> lucky2 = lucky[1:]
>>> lucky2
(7, 21, 77)
>>> lucky
(6, 7, 21, 77)
```

从积极的角度看，不可变性杜绝了无意间修改元组的可能性，有助于防范错误；从消极的角度看，即便对元组做细微的修改，都必须复制整个元组，这增加了修改大型元组所需的时间和内存。如果你发现自己需要频繁地修改某个元组，就应使用列表替代它。

7.3.2 元组函数

表 7-1 列出了最常用的元组函数。相比于字符串和列表，元组的函数较少。下面几个示例演示了如何使用这些函数：

表 7-1 元组函数

函 数 名	返 回 值
x in tup	如果 x 是元组 tup 的一个元组，则返回 True，否则返回 False
len(tup)	元组 tup 包含的元素数
tup.count(x)	元素 x 在元组 tup 中出现的次数
tup.index(x)	元组 tup 中第一个元素 x 的索引；如果 x 未包含在元组 tup 中，将引发 ValueError 异常

```
>>> pets = ('dog', 'cat', 'bird', 'dog')
>>> pets
('dog', 'cat', 'bird', 'dog')
>>> 'bird' in pets
True
>>> 'cow' in pets
False
>>> len(pets)
4
>>> pets.count('dog')
2
>>> pets.count('fish')
0
>>> pets.index('dog')
0
>>> pets.index('bird')
2
>>> pets.index('mouse')
Traceback (most recent call last):
    File "<pyshell#41>", line 1, in <module>
    pets.index('mouse')
ValueError: tuple.index(x): x not in list
```

与字符串一样，也可使用+和*来拼接元组：

```
>>> tup1 = (1, 2, 3)
>>> tup2 = (4, 5, 6)
>>> tup1 + tup2
(1, 2, 3, 4, 5, 6)
>>> tup1 * 2
(1, 2, 3, 1, 2, 3)
```

7.4 列表

列表与元组基本相同，但有一个重要差别：列表是可变的。换句话说，可在不复制的情况下，添加、删除或修改列表元素。实际上，列表比元组用得多得多（有些 Python 程序员只是隐约知道有元组这么回事）。

列表用方括号括起，其中的元素用逗号分隔。与字符串和元组一样，你可使用 len 轻松地获悉列表的长度，还可使用 + 和 * 轻松地拼接列表：

```
>>> numbers = [7, -7, 2, 3, 2]
>>> numbers
[7, -7, 2, 3, 2]
>>> len(numbers)
5
>>> numbers + numbers
[7, -7, 2, 3, 2, 7, -7, 2, 3, 2]
>>> numbers * 2
[7, -7, 2, 3, 2, 7, -7, 2, 3, 2]
```

另外，与字符串和元组一样，你可使用索引和切片来访问元素和子列表：

```
>>> lst = [3, (1,), 'dog', 'cat']
>>> lst[0]
3
>>> lst[1]
(1,)
>>> lst[2]
'dog'
>>> lst[1:3]
[(1,), 'dog']
>>> lst[2:]
['dog', 'cat']
>>> lst[-3:]
[(1,), 'dog', 'cat']
>>> lst[:-3]
[3]
```

请注意，列表可包含任何类型的值：数字、字符串甚至其他序列。空列表用 [] 表示，而只包含一个元素（x）的单元素列表写做 [x]（与元组不同，对单元素列表来说，并非必须以逗号结尾）。

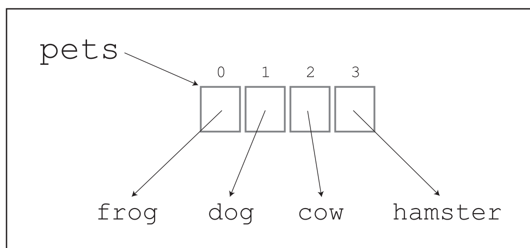


图 7-1 Python 列表元素指向其值

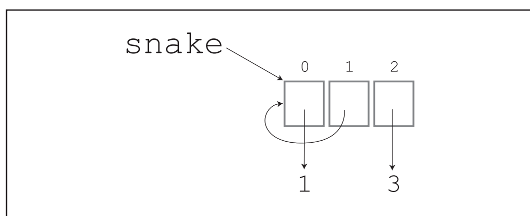


图 7-2 一个自引用列表。请注意，第二个元素指向整个列表，而不是第一个元素

> **术语说明**
>
> 很多 Python 程序员谈论列表时，都给人以列表元素包含其值的感觉。虽然这从技术上说不准确，但是一种常用的便利措词。在处理列表的程序中查找错误时，通常必须明白列表元素指向其值，而不包含它们。

列表是可变的

前面说过，列表是可变的，这是区分列表和元组的重要特征。例如：

```
>>> pets = ['frog', 'dog', 'cow', 'hamster']
>>> pets
['frog', 'dog', 'cow', 'hamster']
>>> pets[2] = 'cat'
>>> pets
['frog', 'dog', 'cat', 'hamster']
```

正如你看到的，这里设置列表 `pets` 的第三个元素，使其指向 `'cat'`。字符串 `'cow'` 被替换掉，并被 Python 自动删除。

图 7-1 图示了列表，对理解列表很有帮助。与变量一样，列表元素指向（而不是包含）相应的值，明白这一点很重要。

列表元素指向相应的值，这可能导致一些不可思议的行为。请看下面的示例：

```
>>> snake = [1, 2, 3]
>>> snake[1] = snake
>>> snake
[1, [...], 3]
```

这里让一个列表元素指向列表本身，创建了一个自引用的数据结构。打印输出中的 `[...]` 表明，Python 能够识别自引用，没有愚蠢地不断打印列表。图 7-2 图示了列表 snake。

7.5 列表函数

列表有很多实用函数，如表 7-2 所示。除 count 只是返回一个数字外，其他所有函数都修改传递给它们的列表。因此它们是修改函数，使用时必须小心。例如，一不小心就会删除错误的元素或在错误的位置插入元素。

函数 append 在列表末尾添加一个元素。一种常见的编程模式是，在函数开头创建一个空列表，然后给列表添加元素。例如，下面的函数根据传入的数字列表创建一个消息列表：

```
# numnote.py
def numnote(lst):
  msg = []
  for num in lst:
    if num < 0:
      s = str(num) + ' is negative'
    elif 0 <= num <= 9:
      s = str(num) + ' is a digit'
    msg.append(s)
  return msg
```

下面是一个调用该函数的示例：

```
>>> numnote([1, 5, -6, 22])
['1 is a digit', '5 is a digit',
'-6 is negative']
```

表 7-2　列表函数

函　数　名	返　回　值
s.append(x)	在列表 s 末尾添加元素 x
s.count(x)	返回元素 x 在列表 s 中出现的次数
s.extend(lst)	将 lst 的所有元素都添加到列表 s 末尾
s.index(x)	返回第一个 x 元素的索引
s.insert(i, x)	将元素 x 插入到索引 i 指定的元素前面，结果是 s[i] == x
s.pop(i)	删除并返回 s 中索引为 i 的元素
s.remove(x)	删除 s 中的第一个 x 元素
s.reverse()	反转 s 中元素的排列顺序
s.sort()	将 s 的元素按升序排列

术语说明

在计算机编程中，术语弹出（pop）通常指的是删除列表的最后一个元素。与之相关的术语是压入（push），指的是在末尾添加一个元素（这正是 Python 函数 append 的功能）。对同一个列表执行弹出和压入时，通常将这个列表称为栈：我们说元素被压入栈顶，然后从栈顶弹出。栈虽然很简单，却是众多高级编程行为（如递归和撤销）的基础。

要打印这些消息，并让每条消息占据一行，可以这样做：

```
>>> for msg in numnote([1, 5, -6, 22]):
→ print(msg)
1 is a digit
5 is a digit
-6 is negative
```

也可以这样做：

```
>>> print('\n'.join(numnote([1, 5, -6, 22])))
1 is a digit
5 is a digit
-6 is negative
```

函数 extend 类似于 append，但在末尾添加一个序列：

```
>>> lst = []
>>> lst.extend('cat')
>>> lst
['c', 'a', 't']
>>> lst.extend([1, 5, -3])
>>> lst
['c', 'a', 't', 1, 5, -3]
```

函数 pop 删除并返回给定索引对应的元素，如下所示：

```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst.pop(2)
'c'
>>> lst
['a', 'b', 'd']
>>> lst.pop()
'd'
>>> lst
['a', 'b']
```

请注意，如果没有向函数 pop 传递索引，它将删除并返回列表末尾的元素。

函数 remove(x) 删除列表中的第一个 x 元素，但不返回该元素：

```
>>> lst = ['a', 'b', 'c', 'a']
>>> lst.remove('a')
>>> lst
['b', 'c', 'a']
```

顾名思义，函数 reverse 反转列表的元素排列顺序：

```
>>> lst = ['a', 'b', 'c', 'a']
>>> lst
['a', 'b', 'c', 'a']
>>> lst.reverse()
>>> lst
['a', 'c', 'b', 'a']
```

函数 reverse 不会制作列表的拷贝，而是直接删除列表中的元素，因此我们说反转是就地完成的。明白这一点很重要。

术语说明

Python 对序列进行排序的方式称为字典顺序（lexicographical ordering）。这是一个通用术语，指的是“字母顺序”，但适用于任何可排序的值序列，而不仅仅是字母。其基本理念是，首先按第一项对元素排序，如果第一项相同，则按第二项进行排序，如果第二项也相同，则按第三项进行排序，依此类推。

7.6 列表排序

对数据进行排序是计算机最常做的事情之一。无论是对人还是计算机来说，排序后的数据通常比未排序的数据更容易处理。例如，在列表中查找最小的元素时，如果列表经过了排序，就根本不需要查找：第一个元素就是。人类通常喜欢排列有序的数据，只要想象一下不按字母顺序印刷的电话簿就明白了！

在 Python 中，要对列表进行排序，最简单的方式是使用函数 sort()。实际上，这个函数可用于对包含数万个元素的列表进行快速排序。与 reverse() 一样，sort() 也就地修改列表：

```
>>> lst = [6, 0, 4, 3, 2, 6]
>>> lst
[6, 0, 4, 3, 2, 6]
>>> lst.sort()
>>> lst
[0, 2, 3, 4, 6, 6]
```

函数 sort 总是按升序排列元素——从最小到最大。如果要按相反的顺序来排列元素——从最大到最小，则最简单的方法是在调用 sort 后再调用 reverse：

```
>>> lst = ['up', 'down', 'cat', 'dog']
>>> lst
['up', 'down', 'cat', 'dog']
>>> lst.sort()
>>> lst
['cat', 'dog', 'down', 'up']
>>> lst.reverse()
>>> lst
['up', 'down', 'dog', 'cat']
```

Python 还知道如何给包含元组的列表排序，如下所示：

```
>>> pts = [(1, 2), (1, -1), (3, 5), (2, 1)]
>>> pts
[(1, 2), (1, -1), (3, 5), (2, 1)]
>>> pts.sort()
>>> pts
[(1, -1), (1, 2), (2, 1), (3, 5)]
```

给列表中的元组排序时，首先按元组的第一个元素排序，如果第一个元素相同，则按第二个元素排序，依此类推。

7.7　列表解析

列表用得如此之多，以至于 Python 提供了一种用于创建列表的特殊表示法——列表解析。下例演示了如何使用列表解析来创建一个由 1~10 的平方组成的列表：

```
>>> [n * n for n in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

这种表示法的主要优点是简洁易读。下面的代码没有使用列表解析，但与前面的代码等价，请对它们进行比较：

```
result = []
for n in range(1, 11):
  result.append(n * n)
```

一旦你掌握了列表解析，就会发现它们编写起来快速而容易，且有众多用途。

7.7.1 列表解析示例

再来看几个列表解析示例。要将列表中的每个数字翻倍并加上 7，可以这样做：

```
>>> [2 * n + 7 for n in range(1, 11)]
[9, 11, 13, 15, 17, 19, 21, 23, 25, 27]
```

要创建一个列表，它包含前 10 个自然数的立方，可这样做：

```
>>> [n ** 3 for n in range(1, 11)]
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

你还可以在列表解析中使用字符串，如下所示：

```
>>> [c for c in 'pizza']
['p', 'i', 'z', 'z', 'a']
>>> [c.upper() for c in 'pizza']
['P', 'I', 'Z', 'Z', 'A']
```

列表解析的一种常见用途是，以某种方式修改现有列表，如下所示：

```
>>> names = ['al', 'mei', 'jo', 'del']
>>> names
['al', 'mei', 'jo', 'del']
>>> cap_names = [n.capitalize() for n in names]
>>> cap_names
['Al', 'Mei', 'Jo', 'Del']
>>> names
['al', 'mei', 'jo', 'del']
```

7.7.2 使用列表解析进行筛选

列表解析还可用于剔除不想要的元素，例如，下面的列表解析返回一个列表，该列表只包含列表 nums 中的正数：

```
>>> nums = [-1, 0, 6, -4, -2, 3]
>>> result = [n for n in nums if n > 0]
>>> result
[6, 3]
```

下面的代码与上述代码等价，但没有使用列表解析：

```
result = []
nums = [-1, 0, 6, -4, -2, 3]
for n in nums:
  if n > 0:
    result.append(n)
```

同样，相比于常规循环，列表解析更简洁易读。

下面的函数使用列表解析删除字符串中的所有元音：

```
# eatvowels.py
def eat_vowels(s):
  """ Removes the vowels from s.
  """
  return ''.join([c for c in s if c.lower()
  → not in 'aeiou'])
```

其效果如下：

```
>>> eat_vowels('Apple Sauce')
'ppl Sc'
```

乍一看，eat_vowels 的函数体晦涩难懂，理解该函数的诀窍是每次阅读一部分。先来看列表解析：

```
[c for c in s if c.lower() not in 'aeiou']
```

这是一个筛选型列表解析，它以每次一个字符的方式扫描 s，将每个字符转换为小写，再检查它是不是元音。如果是元音，则不将其加入最终的列表，否则将其加入最终列表。

该列表解析的结果是一个字符串列表，因此我们使用 join 将所有字符串拼接成一个，再返回这个字符串。

7

生成器表达式

可进一步简化函数 eat_vowels：删除列表解析中的方括号：

```
' '.join(c for c in s if c.lower() not in 'aeiou')
```

这样，传递给 join 的表达式将是一个生成器表达式。在较复杂的 Python 编程中，可使用生成器表达式高效地生成所需的列表或序列部分：根据需要生成元素，而是不像列表解析那样一次性生成所有元素。

7.8 字典

在存储键–值对方面，Python 字典是一种效率极高的数据结构。例如：

```
>>> color = {'red' : 1, 'blue' : 2, 'green' : 3}
>>> color
{'blue': 2, 'green': 3, 'red': 1}
```

字典 color 包含 3 个成员。一个是 'blue':2，其中 'blue' 为键，而 2 是与之关联的值。

要访问字典中的值，可使用相应的键：

```
>>> color['green']
3
>>> color['red']
1
```

使用键来访问字典值的效率极高，即便字典包含数千个键–值对。

与列表一样，字典也是可变的：可以添加或删除键–值对。例如：

```
>>> color = {'red' : 1, 'blue' : 2, 'green' : 3}
>>> color
{'blue': 2, 'green': 3, 'red': 1}
>>> color['red'] = 0
>>> color
{'blue': 2, 'green': 3, 'red': 0}
```

术语说明

字典也称为关联数组、映射或散列表。

散列

Python 字典利用了一个巧妙的编程诀窍——散列。从本质上说，字典中的每个键都被转换为一个数字——散列值，这是使用专门设计的散列函数完成的。字典的值存储在一个底层列表中，并将其散列值用作索引。访问值时，把提供的键转换为散列值，再跳到列表的相应位置。计算散列的细节很复杂，所幸的是，Python 为我们处理了这一切。

7.8.1 对键的限制

你需要知道的是，对字典键有两个限制。首先，字典中的键必须是独一无二的，即在同一个字典中，任何两个键–值对的键都不能相同。例如：

```
>>> color = {'red' : 1, 'blue' : 2, 'green'
→ : 3, 'red' : 4}
>>> color
{'blue': 2, 'green': 3, 'red': 4}
```

虽然我们使用了 `'red'` 键两次，但 Python 只存储了第二个键–值对——`'red':4`。根本不可能使用相同的键：字典键必须是独一无二的。

对键的第二个限制是，键必须是不可变的。因此，字典键不能是列表，也不能是字典。为何要这样要求呢？因为键–值对在字典中的存储位置是根据键计算得到的。即便键发生细微变化，键–值对在字典中的位置也将变化。这可能导致键–值对丢失或无法访问。

这两个限制都不适用于值。值可以是可变的，而相同的值可在同一个字典中出现多次。

7.8.2 字典函数

表 7-3 列出了适用于所有字典的函数。

正如你看到的，要获取字典中的值，标准方式是使用方括号表示法：d[key] 返回与 key 相关联的值。调用 d.get(key) 可完成同样的任务。无论采取哪种方式，如果 key 没有包含在字典 d 中，都将引发 KeyError 异常。

如果你预先不确定某个键是否包含在字典中，可使用 key in d 进行检查。如果 key 包含在字典 d 中，这个表达式将返回 True，否则返回 False。这种检查的效率极高，与用于序列的 in 相比尤其如此。

还可使用函数 pop(key) 和 popitem() 来获取字典值，pop(key) 和 get(key) 之间的差别在于，pop(key) 返回与 key 相关联的值，并将该键 – 值对从字典中删除，而 get 只返回值。

表 7-3 字典函数

函 数 名	返回的值
d.items()	返回一个由字典 d 的键 – 值对组成的视图（view）
d.keys()	返回一个由字典 d 的键组成的视图
d.values()	返回一个由字典 d 的值组成的视图
d.get(key)	返回与 key 相关联的值
d.pop(key)	删除键 key 并返回与之相关联的值
d.popitem()	删除字典 d 中的某个键 – 值对并返回相应的键 – 值
d.clear()	删除字典 d 的所有元素
d.copy()	复制字典 d
d.fromkeys(s, t)	创建一个新字典，其中的键来自 s，值来自 t
d.setdefault(key, v)	如果键 key 包含在字典 d 中，则返回其值；否则，返回 v 并将 (key, v) 添加到字典 d 中
d.update(e)	将 e 中的键 – 值对添加到字典 d 中；e 可能是字典，也可能是键 – 值对序列

函数 popitem() 返回并删除字典的某个键 – 值对，具体是哪个你预先并不知道，因此仅当你不在乎字典元素的访问顺序时，这个函数才适用。

函数 items()、keys() 和 values() 都返回一个特殊对象——视图。视图被链接到原始字典，因此如果字典发生变化，视图也将相应地变化，如下所示：

```
>>> color
{'blue': 2, 'orange': 4, 'green': 3, 'red': 0}
>>> k = color.keys()
>>> for i in k: print(i)
blue
orange
green
red
>>> color.pop('red')
0
>>> color
{'blue': 2, 'orange': 4, 'green': 3}
>>> for i in k: print(i)
blue
orange
green
```

7

7.9 集合

在 Python 中，集合是一系列不重复的元素。集合类似于字典，但只包含键，而没有相关联的值。

集合分两类：可变集合和不可变集合。对于可变集合，你可添加和删除元素，而不可变集合一旦创建就不能修改。

集合最常见的用途可能是用于删除序列中的重复元素，如下所示：

```
>>> lst = [1, 1, 6, 8, 1, 5, 5, 6, 8, 1, 5]
>>> s = set(lst)
>>> s
{8, 1, 5, 6}
```

与字典一样，集合的元素排列顺序也是不确定的。

要获悉所有集合都支持的函数，可在交互式命令行中调用 `dir(set)`，你将发现这样的函数非常多！由于集合用得没有列表和字典那么多，所以这里不列举这些函数。但别忘了，需要使用集合时，可参阅在线文档，网址为 http://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset。

集合与字典

在 Python 中，集合是相对较新的功能。在 Python 还不支持集合时，程序员使用字典来模拟集合，实际上 Python 的第一个集合实现也是这样做的。如果使用了字典，但不关心其值，那么转而使用集合或许可提高代码的可读性。

8

输入和输出

程序要有所作为，就需要与周遭世界通信。它需要与用户交互、读写文件、访问网页等。通常，我们称之为输入和输出（简称I/O）。

你见识过基本的控制台I/O，这包括打印消息以及使用函数input读取用户输入的字符串。本章首先介绍一些设置字符串格式的方法，让你能够通过控制台I/O输出美观的字符串。

接下来，我们将把注意力转向文件I/O，即读写文件。Python提供了强大的基本文件I/O支持，最大限度地简化了程序员的工作。具体地说，我们将介绍如何使用文本文件、二进制文件以及功能强大的pickle模块。

本章内容

- ❑ 设置字符串的格式
- ❑ 格式字符串
- ❑ 读写文件
- ❑ 检查文件和文件夹
- ❑ 处理文本文件
- ❑ 处理二进制文件
- ❑ 读取网页

8.1 设置字符串格式

Python 提供了很多设置字符串格式的方式，这里将讨论较老的字符串插入和较新的格式字符串。

8.1.1 字符串插入

字符串插入是一种设置字符串格式的简单方法，是 Python 从编程语言 C 那里借鉴而来的。例如，下面的示例演示了如何控制小数位数：

```
>>> x = 1/81
>>> print(x)
0.0123456790123
>>> print('value: %.2f' % x)
value: 0.01
>>> print('value: %.5f' % x)
value: 0.01235
```

字符串插入表达式总是采用这样的格式：`format % values`，其中 `format` 是包含一个或多个 % 字符的字符串。在示例 `'value: %.2f' % x` 中，子串 `%.2f` 是一个格式设置命令，让 Python 获取后面提供的第一个值（`x`），并将其显示为包含两位小数的浮点数。

表 8-1　一些转换说明符

说　明　符	含　　义
d	整数
o	八进制（基数为 8 的）值
x	小写十六进制（基数为 16 的）数
X	大写十六进制（基数为 16 的）数
e	小写科学记数法表示的浮点数
E	大写科学记数法表示的浮点数
F	浮点数
s	字符串
%	% 字符

八进制和十六进制

转换说明符 o 和 x 分别将值转换为八进制和十六进制，它们看起来没什么价值，但在很多计算机应用程序中，需要将值表示为十六进制或不那么常见的八进制。正如你在本章后面将看到的，处理二进制文件时经常使用十六进制。

8.1.2　转换说明符

在前面的格式字符串中，f 是一个转换说明符，告诉 Python 如何显示相应的值。表 8-1 列出了最常用的转换说明符。

说明符 e、E 和 f 让你能够以不同的方式表示浮点数。例如：

```
>>> x
0.012345679012345678
>>> print('x = %f' % x)
x = 0.012346
>>> print('x = %e' % x)
x = 1.234568e-02
>>> print('x = %E' % x)
x = 1.234568E-02
```

你可根据需要在格式字符串中包含任意数量的说明符，但必须为每个说明符提供一个值。例如：

```
>>> a, b, c = 'cat', 3.14, 6
>>> s = 'There\'s %d %ss older than %.2f
→ years' % (c, a, b)
>>> s
"There's 6 cats older than 3.14 years"
```

从这个示例可知，格式字符串相当于简单模板，将用指定的值填充。值是以元组方式指定的，它们的排列顺序必须与替换顺序一致。

提 示　转换说明符 d、f 和 s 最常用，因此你有必要记住它们，尤其是 f，它是控制浮点数格式的最简单方式。

提 示　如果要在字符串中包含字符 %，必须使用 '%%'。

8.2 格式字符串

在 Python 中，另一种创建美观字符串的方式是结合使用格式字符串和字符串函数 format(value, format_spec)。例如：

```
>>> 'My {pet} has {prob}'.format
➝ (pet = 'dog', prob='fleas')
'My dog has fleas'
```

在格式字符串中，用大括号括起的内容都将被替换，这称为命名替换（named replacement）。就这个示例而言，命名替换的可读性极佳。

你还可以按位置替换值：

```
>>> 'My {0} has {1}'.format ('dog', 'fleas')
'My dog has fleas'
```

还可以像字符串插入那样使用转换说明符：

```
>>> '1/81 = {x}'.format(x=1/81)
'1/81 = 0.0123456790123'
>>> '1/81 = {x:f}'.format(x=1/81)
'1/81 = 0.012346'
>>> '1/81 = {x:.3f}'.format(x=1/81)
'1/81 = 0.012'
```

模板包

在字符串插入和格式字符串都不够强大或灵活时，可能需要使用模板包，如 Cheetah（www.cheetahtemplate.org）或 Django（www.djangoproject.com）提供的模板包。这两个模板包都让你能够完成非常复杂的替换，在需要创建大量动态生成的网页时，这是不错的选择。

你可以使用大括号来指定格式设置参数，如下所示：

```
>>> 'num = {x:.{d}f}'.format (x=1/81, d=3)
'num = 0.012'
>>> 'num = {x:.{d}f}'.format (x=1/81, d=4)
'num = 0.0123'
```

使用常规的字符串插入无法做到这一点。

提 示 如果要在格式字符串中表示字符 { 或 }，可使用 {{ 和 }}。

提 示 相比字符串插入，格式字符串更灵活、更强大，但也更复杂。如果只想创建一些格式简单的字符串，字符串插入可能是最佳的选择；而格式字符串更适合庞大而复杂的设置任务，如创建网页或格式邮件。

8.3　读写文件

文件是一个命名的比特集合，存储在硬盘、U 盘、闪存条等辅助存储设备中。这里将文件分为两类：文本文件和二进制文件，其中前者本质上是存储在磁盘中的字符串，而后者是其他各种内容。

文本文件具有如下特点。

- 基本上是磁盘中的字符串。Python 源代码文件和 HTML 文件都属于文本文件。
- 可使用任何文本编辑器进行编辑，因此对人类来说相对容易阅读和修改。
- 对程序来说，它们通常难以阅读。通常，每种文本文件都需使用相应的分析程序（parser）来阅读，例如，Python 使用专用分析程序来帮助阅读 .py 文件，而要阅读 HTML 文件，需要使用专用于 HTML 的分析程序。
- 通常比等价的二进制文件大。需要通过网络发送大型文本文件时，这是个严重的问题。因此，通常对文本文件进行压缩（如压缩成 zip 格式），以提高传输速度和节省磁盘空间。

二进制文件具有如下特点。

- 通常是人类无法阅读的，至少使用常规文本编辑器无法查看。在文本编辑器中打开二进制文件时，显示的是一堆乱码。有些类型的二进制文件（如JPEG 图像）需要使用特殊查看器来显示其内容。
- 占据的空间通常比等价的文本文件小。例如，二进制文件可能将其保存的信息编组，每组包含 32 位，而两组之间不使用逗号或空格等分隔符。
- 对程序来说，它们读写起来通常比文本文件容易。虽然二进制文件各不相同，但通常无需编写复杂的分析程序来读取它们。
- 通常与特定程序相关联，如果没有该程序，通常无法使用它们。有些流行的二进制文件的格式是公开的，因此如果你愿意，可以自己编写读写它们的程序，但通常需要花很大的功夫。

8.3.1 文件夹

除文件外，人们还使用文件夹（目录）来存储文件和其他文件夹。大多数文件系统的文件夹结构都庞大而复杂，呈层次型。

路径名是用于标识文件或文件夹的名称。完整的路径名可能非常长，例如，在我的 Windows 计算机上，Python 安装文件夹的完整路径名如下：C:\Documents and Settings\tjd\Desktop\python。

Windows 路径名使用反斜杠（\）来分隔路径中的名称，并以盘符（这里是 C:）打头。

在 Mac 和 Linux 系统中，使用斜杠（/）来分隔名称，且不以盘符打头。例如，在我的 Linux 系统中，Python 安装文件夹的完整路径名如下：/home/tjd/Desktop/python。

提示　本书前面说过，如果要在 Python 字符串中包含 \ 字符，必须使用 \：

```
'C:\\home\\tjd\\Desktop\\python'
```

为避免使用两个反斜杆，可使用原始字符串：

```
r'C:\home\tjd\Desktop\python'
```

提示　在 Python 程序中支持这两种风格的路径名有点棘手，有关这方面的更详细信息，请参阅有关 Python 模块 os.path 的文档。

8.3.2 当前工作目录

很多程序都使用了当前工作目录（cwd）的概念，这指的是默认目录。操作文件或文件夹时，如果你没有提供完整路径名，Python 将假定你指的是当前工作目录中的相应文件或文件夹。

8.4　检查文件和文件夹

Python 提供了很多这样的函数：返回有关计算机文件系统中文件和文件夹的信息。表 8-2 列出了其中最实用的几个。

下面来编写几个函数，看看上述函数的工作原理。例如，一种常见的任务是获悉当前工作目录中的文件和文件夹。代码 os.listdir(os.getcwd()) 比较难看，因此我们编写下面这样的函数：

```
# list.py
def list_cwd():
  return os.listdir(os.getcwd())
```

下面是两个相关的辅助函数，它们使用列表解析分别返回当前工作目录中的文件和文件夹：

```
# list.py
def files_cwd():
  return [p for p in list_cwd()
          if os.path.isfile(p)]
def folders_cwd():
  return [p for p in list_cwd()
          if os.path.isdir(p)]
```


表 8-2　实用的文件和文件夹函数

函　数　名	作　　用
os.getcwd()	返回当前工作目录的名称
os.listdir(p)	返回一个字符串列表，其中包含路径 p 指定的文件夹中所有文件和文件夹的名称
os.chdir(p)	将当前工作目录设置为路径 p
os.path.isfile(p)	当路径 p 指定的是一个文件的名称时，返回 True，否则返回 False
os.path.isdir(p)	当路径 p 指定的是一个文件夹的名称时，返回 True，否则返回 False
os.stat(fname)	返回有关 fname 的信息，如大小（单位为字节）和最后一次修改时间

如果只想获悉当前工作目录中的 .py 文件，可编写如下函数：

```
# list.py
def list_py(path = None):
  if path == None:
    path = os.getcwd()
  return [fname for fname in os.listdir(path)
      if os.path.isfile(fname)
      if fname.endswith('.py')]
```

这个函数巧妙地利用了其输入参数：如果你调用 `list_py()` 时未提供参数，它将把当前工作目录视为目标文件夹，否则将指定的目录视为目标文件夹。

最后，下面的函数返回当前工作目录中所有文件的大小总和：

```
# list.py
def size_in_bytes(fname):
  return os.stat(fname).st_size
def cwd_size_in_bytes():
  total = 0
  for name in files_cwd():
    total = total + size_in_bytes(name)
  return total
```

一个巧妙的诀窍

可重写函数 cwd_size_in_bytes，使其只包含一行代码：

```
def cwd_size_in_bytes2():
  return sum(size_in_bytes(f)
             for f in files_cwd())
```

详细解释 cwd_size_in_bytes2 的工作原理超出了本书的范围，如果你对这种更简洁的方式感兴趣，可在网上搜索 Python 生成器表达式。

提示 为节省篇幅，我们省略了这些函数的文档字符串。然而，在 Google pythonintro 网站（http://pythonintro.googlecode.com）提供的示例代码文件中，都包含文档字符串。

提示 从名称 `cwd_size_in_bytes` 可知，其返回值以字节为单位。通过在函数名中包含返回值的单位，无需查看文档就能获悉单位。

提示 一般而言，大量使用函数是个不错的主意。即便函数像 `list_dir()` 那样只有一行代码，也很有帮助，因为它们让程序更容易理解和维护。

提示 函数 `os.stat()` 非常复杂，它提供的有关文件的信息比这里演示的多得多。有关这个函数的更详细信息，请参阅 Python 在线文档（http://docs.python.org/3/library/os.html）。

8.5　处理文本文件

在 Python 中，处理文本文件相对容易。通常采用图 8-1 所示的 3 个步骤来处理文件。

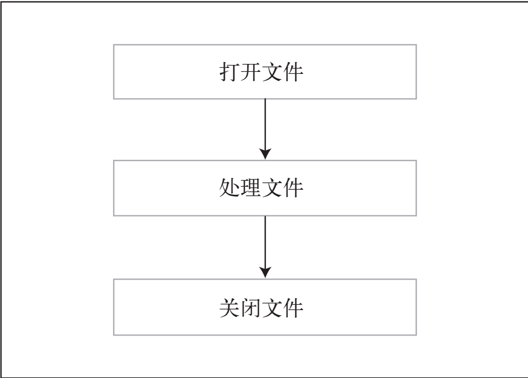


图 8-1　文本文件的 3 个主要处理步骤。要使用文件，必须先打开它；使用完毕后再将其关闭，确保将所有修改都提交给文件

8.5.1　逐行读取文本文件

读取文本文件的最常见方式可能是每次读取一行。例如，下面的代码在屏幕上打印文件的内容：

```
# printfile.py
def print_file1(fname):
  f = open(fname, 'r')
  for line in f:
    print(line, end = '')
  f.close() \# 这行代码是可选的
```

这个函数的第 1 行打开指定的文件：调用函数 open 时，必须指定你要处理的文件的名称，还必须指定打开模式。这里只读取文件，因为以读取模式（'r'）打开它。表 8-3 列出了 Python 中主要的文件打开模式。函数 open 返回一个特殊的文件对象，表示磁盘中的文件。最重要的是，open 不将文件读取到内存中。在上述示例中，使用 for 循环以每次一行的方式读取文件。函数 print_file1 的最后一行关闭文件；正如注释指出的，这是可选的，因为 Python 几乎总是会自动为你关闭文件。在这里，f 是函数 print_file1 中的一个局部变量，因为函数 print_file1 结束时，Python 将自动关闭并删除 f 指向的文件对象（不是文件本身）。

表 8-3　Python 文件打开模式

字　符	含　义
'r'	为读取而打开文件（默认模式）
'w'	为写入而打开文件
'a'	为在文件末尾附件而打开文件
'b'	二进制模式
't'	文本模式（默认模式）
'+'	为读写打开文件

默认为读取模式

如果你只想读取文本文件，可在调用函数 open 时只传递文件名。例如：

```
f = open(fname)
```

未指定模式参数时，Python 假定你要为读取而打开文本文件。

提 示 在函数 print_file1 中，print 语句将 end 设置为 ''，因为文件中的各行都以字符 \n 结尾。如果将该 print 语句改为 print(line)，显示的文件内容中将包含额外的空行（试试就明白了）。

提 示 如果打开文件时出错，程序可能在未妥善关闭文件的情况下终止。下一章将介绍如何处理这样的错误，确保总是妥善地关闭文件。

8.5.2 将整个文本文件作为一个字符串进行读取

另一种读取文本文件的常见方式是，将其作为一个大型字符串进行读取，如下所示：

```
# printfile.py
def print_file2(fname):
  f = open(fname, 'r')
  print(f.read())
  f.close()
```

这比函数 print_file1 短小、简单，很多程序员都喜欢采用这种方式。然而，如果要读取的文件非常大，将占用大量内存，这可能降低计算机的运行速度，甚至导致计算机崩溃。

最后，我们注意到很多程序员将上述函数编写为下面一行代码：

```
print(open(fname, 'r').read())
```

这种形式更紧凑，可能需要一段时间才能习惯，但很多程序员都喜欢这种风格，因为其输入量少，可读性也相对较高。

8.5.3　写入文本文件

写入文本文件只比读取文本文件复杂一点点，例如，下面这个函数新建一个名为 story.txt 的文本文件：

```
# write.py
def make_story1():
  f = open('story.txt', 'w')
  f.write('Mary had a little lamb,\n')
  f.write('and then she had some more.\n')
```

'w' 让 Python 以写入模式打开文件。为将文本写入文件，你调用 f.write，并将要写入的字符串传递给它。字符串将以指定的顺序写入文件。

需要注意的是，如果文件 story.txt 已经存在，调用 open('story.txt', 'w') 将删除它！如果你不想覆盖 story.txt，应先检查它是否存在：

```
# write.py
import os
def make_story2():
  if os.path.isfile('story.txt'):
    print('story.txt already exists')
  else:
    f = open('story.txt', 'w')
    f.write('Mary had a little lamb,\n')
    f.write('and then she had some more.\n')
```

8.5.4　附加到文本文件末尾

将字符串加入到文本文件时，一种常见的方式是将它们附加到文件末。与模式 'w' 不同,这种模式不会删除文件既有的内容。例如：

```
def add_to_story(line, fname = 'story.txt'):
  f = open(fname, 'a')
  f.write(line)
```

这里需要注意的一个要点是，文件是以附加模式（'a'）打开的。

8.5.5　将字符串插入到文件开头

相比于在文件末尾附加字符串，将字符串写入文件开头不那么容易，因为操作系统 Windows、Linux 和 Macintosh 都没有为这样做提供直接支持。要将文本插入到文件开头，最简单的方式可能如下：将文件读取到一个字符串中，将新文本插入到该字符串，再将这个字符串写入原来的文件，如下所示：

```
def insert_title(title, fname = 'story.txt'):
  f = open(fname, 'r+')
  temp = f.read()
  temp = title + '\n\n' + temp
  f.seek(0) \# 让文件指针指向文件开头
  f.write(temp)
```

首先，注意到我们使用了特殊模式 'r+' 来打开文件，这意味着可读取和写入文件。接下来，我们将整个文件读取到字符串变量 temp 中，并使用字符串拼接插入标题（title）。

将新创建的字符串写回文件前，必须先让文件对象 f 重置其内部的文件指针。所有文本文件对象都记录了它当前指向文件的什么位置，调用 f.read() 后，文件指针指向文件末尾。通过调用 f.seek(0)，让文件指针重新指向了文件开头，这样写入 f 时，将从文件开头开始。

8

8.6 处理二进制文件

如果文件不是文本文件，它就被视为二进制文件。二进制文件以模式 'b' 打开，而你可访问其各个字节。例如：

```
def is_gif(fname):
  f = open(fname, 'br')
  first4 = tuple(f.read(4))
  return first4 == (0x47, 0x49, 0x46, 0x38)
```

这个函数检查 fname 是不是 GIF 图像文件，方法是检查其前 4 个字节是不是 (0x47, 0x49, 0x46, 0x38)（所有 GIF 图像文件都以这 4 个字节打头）。

在 Python 中，类似于 0x47 的数字为十六进制数。十六进制数非常适合用于处理字节，因为每个十六进制位对应于 4 比特，因此使用两个十六进制位（0x47）可描述一个字节。

注意到文件是以 'br' 模式打开的，这表示二进制读取模式。读取二进制文件时，调用 f.read(n) 来读取接下来的 n 个字节。与文本文件对象一样，二进制文件对象也使用文件指针来记录接下来应读取文件的哪个字节。

术语说明

Python 模块 pickle 执行的操作通常被称为对象串行化（简称串行化）。其基本思想是，将复杂的数据结构转换为字节流，即创建数据结构的串行化表示。

pickle

访问二进制文件的各个字节是一种非常低级的操作，虽然在系统编程中很有用，但在较高级的应用程序编程中用途有限。

在处理二进制文件方面，pickle 通常是一种方便得多的方式。Python 模块 pickle 让你能够轻松地读写几乎任何数据结构，如下所示：

```
# picklefile.py
import pickle
def make_pickled_file():
  grades = {'alan' : [4, 8, 10, 10],
            'tom' : [7, 7, 7, 8],
            'dan' : [5, None, 7, 7],
            'may' : [10, 8, 10, 10]}
  outfile = open('grades.dat', 'wb')
  pickle.dump(grades, outfile)

def get_pickled_data():
  infile = open('grades.dat', 'rb')
  grades = pickle.load(infile)
  return grades
```

基本上，你可使用 `pickle.dump` 将数据结构存储到磁盘，以后再使用 `pickle.load` 从磁盘获取数据结构。对很多程序来说，这都是一项特别有用的功能，因此每当需要存储二进制数据时，都应考虑使用这种功能。

提示 除数据结构外，还可使用 pickle 来存储函数。

提示 pickle 不能用于读写特殊格式的二进制文件，如 GIF 文件。对于这样的文件，必须逐字节处理。

提示 Python 包含一个名为 shelve 的模块，这个模块提供了存储和检索数据的更高级方式。模块 shelve 让你能够将文件视为字典，更详细的信息请参阅 Python 文档（http://docs.python.org/3/library/shelve.html）。

提示 Python 还有一个名为 sqlite3 的模块，这个模块提供了访问 SQLite 数据库的接口，让你能够编写 SQL 命令来存储和检索数据，就像使用 Postgres 和 MySQL 等大型数据库产品时一样。有关这方面的更详细信息，请参阅 Python 文档（http://docs.python.org/3/library/sqlite3.html）。

8.7 读取网页

Python 为访问网络提供了强大支持。一种常见的任务是让程序自动读取网页，而使用模块 urllib 可轻松地完成这种任务：

```
>>> import urllib.request
>>> page = urllib.request. urlopen
→ ('http://www.python.org')
>>> html = page.read()
>>> html[:25]
b'<!DOCTYPE html PUBLIC "-/'
```

html 包含 www.python.org 处网页的全部文本。这些文本是 HTML 格式的，因此与你在 Web 浏览器中使用“查看源代码”选项看到的结果一样。将网页作为字符串存储到计算机中后，便可使用 Python 字符串操作函数提取其中的信息。

提示　模块 urllib 还让你能够以编程方式发送数据，更详细信息请参阅 Python 文档（http://docs.python.org/3/howto/urllib2.html）。

提示　要将网页读取到字符串变量中，第一步是创建一个 Web 请求。接下来的一个重要步骤是分析字符串——识别并提取标题、段落、表格等。Python 通过模块 html.parser 提供了一个基本的 HTML 分析库，更详细的信息请参阅 Python 文档（http://docs.python.org/3/library/html.parser.html）。

提示　另一个绝妙的模块是 webbrowser，它让你能够以编程方式在浏览器中显示网页。例如，下述代码在默认 Web 浏览器中显示雅虎的主页：

```
>>> import webbrowser
>>> webbrowser.open ('http://www.yahoo.com')
True
>>>
```

9

异常处理

程序如何处理意料之外的错误呢？异常为这个难题提供了解决方案。例如，如果在读取文件期间，计算机上的其他程序将其删除了，结果将如何呢？如果程序从网站下载网页时，该网站突然崩溃，结果又将如何呢？

在这些以及众多其他情形下，Python 采取的措施是引发异常。异常是一种特殊的错误对象，你可以捕获并检查它们，以决定如何处理错误。

异常可能改变程序的控制流程。根据发生的时机，异常可能导致执行流程跳出函数或进入处理错误的代码块。

通常，你无法准确确定哪一行可能引发异常，这带来了一些棘手的问题。因此，Python 提供了一个特殊的异常处理结构，可用于捕获异常，并确保无论是否出现异常都将执行清理代码。

本章内容

- 异常
- 捕获异常
- 清理操作

9.1 异常

一个异常的例子是 IOError，当你试图打开不存在的文件时将引发这种异常：

```
>>> open('unicorn.dat')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
  open('unicorn.dat')
  File "C:\Python30\lib\io.py", line 284,
  → in __new__
  return open(*args, **kwargs)
  File "C:\Python30\lib\io.py", line 223,
  → in open
  closefd)
IOError: [Errno 2] No such file or directory:
→ 'unicorn.dat'
```

出现异常后，如果不捕获或以任何其他方式进行处理，Python 将立即停止运行程序，并显示栈跟踪——异常发生前调用的函数清单。这对确定导致错误的代码行很有帮助。

在上述栈跟踪中，最后一行表明引发了 IOError 异常，具体地说，这意味着在当前工作目录中找不到文件 unicorn.dat。IOError 显示的错误消息随导致异常的原因而异。

术语说明

在 Python 中发生异常时，我们称之为异常被引发或抛出。发生异常时，如果不采取任何措施，程序通常会立即停止运行，并显示栈跟踪。然而，正如你稍后将看到的，在要供他人使用的程序中，通常捕获并处理异常。

引发异常

正如你从函数 open 身上看到的，Python 内置函数和库函数通常在出现意外情况时引发异常。

例如，除以零将抛出异常：

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
  1/0
ZeroDivisionError: int division or modulo
→ by zero
```

在 Python 中，语法错误也会导致异常：

```
>>> x := 5
SyntaxError: invalid syntax (<pyshell#2>,
→ line 1)
>>> print('hello world)
SyntaxError: EOL while scanning string
→ literal (<pyshell#3>, line 1)
```

另外，在代码的任何地方都可使用 raise 语句故意引发异常，如下所示：

```
>>> raise IOError('This is a test!')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
  raise IOError('This is a test!')
IOError: This is a test!
```

Python 包含大量内置的异常，这些异常被组织成层次结构，更详细的信息请参阅 Python 文档（http://docs.python.org/3/library/exceptions.html#bltin-exceptions）。

9.2 捕获异常

异常发生时，你有如下两种选择。

1. 忽略异常，让程序崩溃并显示栈跟踪。在开发程序期间，你通常想这样做，因为栈跟踪提供的调试信息很有帮助。

2. 捕获异常，并打印友好的错误消息乃至试图修复问题。对于要供非程序员使用的程序，几乎都应这样做。普通用户可不想看到栈跟踪！

下面的示例演示了如何捕获异常。这里假设你要从用户那里获取一个整数，为此你反复提示用户，直到用户输入有效的整数：

```
def get_age():
  while True:
    try:
      n = int(input('How old are you? '))
      return n
    except ValueError:
      print('Please enter an integer value.')
```

这个函数中的 while 循环是一个 try/except 块。你可将可能引发异常的代码放在 try 块中。

函数会引发哪些异常

你怎么知道在函数 get_age() 中应检查异常 ValueError 呢？这是通过函数的文档获悉的。文档完备的函数会指出它可能引发哪些异常，例如，函数 open 的文档（http://docs.python.org/3/library/functions.html?#open）指出，它可能引发异常 IOError。然而，并非所有的 Python 内置函数都如此友善，例如，函数 int 的文档就对其可能引发的异常一字未提。在这种情况下，要搞清楚可能出现的异常，必须阅读其他 Python 示例代码或使用命令行进行试验。

只要 try 块中的代码出现异常，就将跳过其他所有未执行的语句，立即跳转到 except 块。在这个示例中，如果出现异常，将跳过 return 语句。

如果 try 块没有引发异常，将忽略（跳过）exceptValueError 块。

在这个示例中，如果用户输入的字符串不是有效的整数，函数 int() 将引发异常 ValueError，进而跳转到 except ValueError 块并打印错误消息。出现 ValueError 异常时，将跳过 return 语句——立即跳转到 except 块。

如果用户输入的是有效整数，就不会引发异常，因此 Python 将接着执行 return 语句，从而结束函数。

9.2.1 try/except 块

try/except 块的工作原理有点像 if 语句，但存在一个重大不同：if 语句根据布尔表达式的结果决定如何做，而 try/except 块根据是否出现了异常决定如何做。

同一个函数可能引发多种异常，还可能出于不同原因引发相同的异常。下面是函数 int() 可能引发的 3 种异常（为方便阅读，删除了栈跟踪）：

```
>>> int('two')
ValueError: invalid literal for int() with
➝ base 10: 'two'
>>> int(2, 10)
TypeError: int() can't convert non-string
➝ with explicit base
>>> int('2', 1)
ValueError: int() arg 2 must be >= 2 and <= 36
```

这表明 int() 至少会出于两个不同的原因引发异常 ValueError，还至少会出于另一个原因引发异常 TypeError。

9.2.2　捕获多种异常

你可编写处理多种异常的 try/except 块。为此，可在 except 子句中指定多种异常：

```
def convert_to_int1(s, base):
  try:
    return int(s, base)
  except (ValueError, TypeError):
    return 'error'
```

如果要分别处理不同的异常，可使用多个 except 子句：

```
def convert_to_int2(s, base):
  try:
    return int(s, base)
  except ValueError:
    return 'value error'
  except TypeError:
    return 'type error'
```

9.2.3　捕获所有异常

如果你在 except 子句中没有指定异常，它将捕获所有异常：

```
def convert_to_int3(s, base):
  try:
    return int(s, base)
  except:
    return 'error'
```

这种 except 子句将捕获所有异常，它不关心发生的是哪种错误，而只关心是否发生了错误。在很多情况下，这就足够了。

9.3 清理操作

在 try/except 块中，可包含执行清理操作的 finally 代码块，如下所示：

```
def invert(x):
  try:
    return 1 / x
  except ZeroDivisionError:
    return 'error'
  finally:
    print('invert(%s) done' % x)
```

finally 块肯定会执行，它要么在执行 try 块后执行，要么在执行 except 块后执行。在不管是否发生异常都要执行某些代码时，这很有用。例如，通常将关闭文件的语句放在 finally 块中，这样文件肯定会被关闭，即便发生了 IOError 异常。

另一种格式设置方式

在上述两个代码片段中，为让文件每行开头的行号右对齐，并在行号左边填充零，print 语句都使用了字符串插入。如果你想使用格式字符串，可将这些 print 语句替换为下述语句：

```
print('{0:04} {1}'.format(num, line),
→ end = '')
```

with 语句

为确保即便发生异常，也将尽早执行清理操作（如关闭文件），还可使用 Python 语句 with。例如，请看下面的代码，它在屏幕上打印文件内容，并给每一行都加上行号：

```
num = 1
f = open(fname)
for line in f:
  print('%04d %s' % (num, line), end = '')
  num = num + 1
  \# 后续代码
```

这里不知道文件对象 f 将在何时关闭。f 通常在 for 循环结束后关闭，但不知道准确的时间。换句话说，不再需要后，f 保持打开状态多长时间是不确定的，如果其他程序试图访问这个文件，这可能是个问题。

为确保不再需要的文件被尽早关闭，可使用 with 语句：

```
num = 1
with open(fname, 'r') as f:
  for line in f:
    print('%04d %s' % (num, line), end = '')
    num = num + 1
```

这个代码片段的屏幕输出与前一个代码片段相同，但使用 with 语句时，将在 for 循环结束后立即执行文件对象清理操作（即关闭文件），避免了不再需要的 f 处于打开状态。

10

面向对象编程

本章简要地介绍面向对象编程（简称OOP）。OOP是一种组织程序的方法，提倡仔细设计和代码重用。大多数现代编程语言都支持OOP，事实证明这是一种组织和创建大型程序的实用方式。

从本质上说，对象是一组数据以及操作这些数据的函数。本书一直在使用Python对象，因为数字、字符串、列表、字典和函数都是对象。

要创建新型对象，必须先创建类。从本质上说，类就是设计蓝图，用于创建特定类型的对象。类指定了对象将包含哪些数据和函数，还指定了对象与其他类的关系。对象封装了数据以及操作这些数据的函数。

一个重要的OOP功能是*继承*：创建新类时，可让它继承既有类的数据和函数。妥善地使用继承可避免重新编写代码，还可让程序更容易理解。

本章内容

- 编写类
- 显示对象
- 灵活的初始化
- 设置函数和获取函数
- 继承
- 多态
- 更深入地学习

10.1 编写类

下面就来介绍 OOP——编写一个表示人的简单类：

```
# person.py
class Person:
  """ Class to represent a person
  """
  def __init__(self):
    self.name = ''
    self.age = 0
```

上述代码定义了一个名为 Person 的类。它定义了 Person 对象包含的数据和函数。Person 类很简单，它包含数据 name 和 age；当前唯一一个函数是 __init__，这是用于初始化对象值的标准函数。正如你将看到的，你创建 Person 对象时，Python 将自动调用 __init__。

在类中定义的函数被称为方法。与 __init__ 一样，方法的第一个参数必须是 self（稍后将更详细地讨论 self）。

可以像下面这样使用 Person 对象：

```
>>> p = Person()
>>> p
<__main__.Person object at
→ 0x00AC3370>
>>> p.age
0
>>> p.name
''
>>> p.age = 55
>>> p.age
55
>>> p.name = 'Moe'
>>> p.name
'Moe'
```

术语说明

在有些 OOP 语言中，__init__ 被称为构造函数，因为它构造对象。每次创建新对象时，都将调用构造函数。在 Java 和 C++ 等语言中，创建对象时需要使用关键字 new。

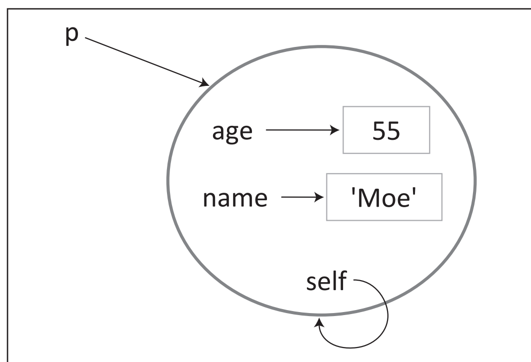


图 10-1　在这个例子中，变量 p 指向一个 Person 对象（由圆圈表示）。从 Person 类的定义可知，Person 对象包含变量 age 和 name，你可以像使用常规变量那样使用它们，但必须使用句点表示法，即 p.age 和 p.name。Python 自动给每个对象添加特殊变量 self，这个变量指向对象本身，让类中的函数能够明确地引用对象的数据和函数

要创建 Person 对象，只需调用 Person()。这导致 Python 运行 Person 类的函数 __init__，并返回一个新的 Person 对象。

变量 age 和 name 包含在对象中，因此每个新创建的 Person 对象都有自己的 age 和 name。要访问 age 和 name，必须使用句点表示法指定存储它们的对象。

参数 self

你可能注意到了，我们调用 Person() 时没有提供任何参数，但函数 __init__(self) 期望获得名为 self 的输入。这是因为在 OOP 中，self 是一个指向对象本身的变量，如图 10-1 所示。这个概念很简单，却难倒了很多初学者。

提示　所有类都应该有方法 __init__(self)，这个方法的职责是初始化对象，如初始化对象的变量。方法 __init__ 仅被调用一次——在对象被创建时。正如你将看到的，可根据需要给 __init__ 指定其他参数。

提示　我们遵循 Python 的标准做法，将 __init__ 的第一个参数命名为 self。并非必须使用这样的名称，你根据自己的喜好使用任何变量名（而不是 self）。然而，使用 self 是大家普遍接受的 Python 约定，如果你使用其他名称，只会让阅读你代码的其他程序员迷惑。Java 和 C++ 等其他一些语言要求必须使用名称 this。

提示　在 Python 中，可像使用其他数据类型一样使用对象：可将它们传递给函数、存储到列表和字典中、保存到文件中，等等。

10.2 显示对象

前面说过，方法是在类中定义的函数。下面给 Person 类添加一个方法，用于打印 Person 对象的内容：

```
# person.py
class Person:
  """ Class to represent a person
  """
  def __init__(self):
    self.name = ''
    self.age = 0
  def display(self):
    print("Person('%s', age)" % (self.name,
    → self.age))
```

方法 display 将 Person 对象的内容以适合程序员阅读的格式打印到屏幕上：

```
>>> p = Person()
>>> p.display()
Person('', 0)
>>> p.name = 'Bob'
>>> p.age = 25
>>> p.display()
Person('Bob', 25)
```

方法 display 的效果很好，但我们还可以做得更好：Python 提供了一些特殊方法，让你能够定制对象以支持天衣无缝的打印。例如，特殊方法 __str__ 用于生成对象的字符串表示：

```
# person.py
class Person:
  # 为节省篇幅，省略了 __init__
  def display(self):
    print("Person('%s', age)" %
    → (self.name, self.age))
  def __str__(self):
    return "Person('%s', age)" %
    → (self.name, self.age)
```

现在我们可以这样编写代码：

```
>>> p = Person()
>>> str(p)
"Person('', 0)"
```

还可使用 str 来简化方法 display：

```
# person.py
class Person:
  # 为节省篇幅，省略了 __init__
  def display(self):
    print(str(self))
  def __str__(self):
    return "Person('%s', age)" %
    → (self.name, self.age)
```

你还可定义特殊方法 __repr__，它返回对象的“官方”（official）表示。例如，Person 对象的默认官方表示不太实用：

```
>>> p = Person()
>>> p
<__main__.Person object at 0x012C3170>
```

通过添加方法 __repr__，我们可控制这里打印的字符串。在大多数类中，方法 __repr__ 都与方法 __str__ 相同：

```
# person.py
class Person:
  # 为节省篇幅，省略了 __init__
  def display(self):
    print(str(self))
  def __str__(self):
    return "Person('%s', age)" %
    → (self.name, self.age)
  def __repr__(self):
    return str(self)
```

现在 Person 对象使用起来更容易：

```
>>> p = Person()
>>> p
Person('', 0)
>>> str(p)
"Person('', 0)"
```

提示 创建自己的类和对象时，编写函数 __str__ 和 __repr__ 几乎总是值得的。它们对于显示对象的内容很有帮助，而显示对象的内容有助于调试程序。

提示 如果你定义了方法 __repr__，但没有定义方法 __str__，则对对象调用 str() 时，将执行 __repr__。

提示 添加方法 __repr__ 后，可进一步简化方法 display：

```
def display(self):
print(self)
```

实际上，通常没有必要编写方法 display。

提示 Python 文档建议将对象的字符串表示设置为创建对象所需的代码。这种约定很有用，让你能够轻松地创建对象——只需将字符串表示复制并粘贴到命令行。

10.3 灵活的初始化

当前，要创建具有特定姓名和年龄的 Person 对象，必须这样做：

```
>>> p = Person()
>>> p.name = 'Moe'
>>> p.age = 55
>>> p
Person('Moe', 55)
```

一种更方便的方法是，在构造对象时将姓名和年龄传递给 __init__。为此，需要重写 __init__：

```
# person.py
class Person:
  def __init__(self, name = '',
               age = 0):
    self.name = name
    self.age = age
```

这样，初始化 Person 对象将简单得多：

```
>>> p = Person('Moe', 55)
>>> p
Person('Moe', 55)
```

由于 __init__ 的参数有默认值，你甚至可以创建“空的”Person 对象：

```
>>> p = Person()
>>> p
Person('', 0)
```

注意方法 __init__，我们在其中使用了 self.name 和 name（以及 self.age 和 age）。变量 name 指向传入 __init__ 的值，而 self.name 指向存储在对象中的值。使用 self 更清楚地指出了谁是谁。

提 示　虽然给 __init__ 的参数指定默认值很容易，进而让你能够轻松地创建空的 Person 对象，但从设计角度看，这是不是个好主意就不那么显而易见了。空的 Person 对象没有真正意义上的姓名和年龄，因此需要在处理 Person 对象的代码中检查这一点。始终需要检查特殊情形很快会成为负担，而且容易忘记。因此，很多程序员选择不像这里那样给 __init__ 的参数指定默认值。

10.4　设置函数和获取函数

当前，我们可以使用句点表示法来读写 Person 对象的 name 和 age 值：

```
>>> p = Person('Moe', 55)
>>> p.age
55
>>> p.name
'Moe'
>>> p.name = 'Joe'
>>> p.name
'Joe'
>>> p
Person('Joe', 55)
```

这种做法存在的一个问题是，可能不小心将年龄设置为荒谬的值，如 –45 或 509。对于常规 Python 变量，无法对可赋给它的值进行限制。但在对象中，可编写特殊的设置函数（setter）和获取函数（getter），对存取值的方式进行控制。

首先，添加一个设置函数，它仅在提供的值合理时才修改 age：

```
def set_age(self, age):
  if 0 < age <= 150:
    self.age = age
```

装饰器

装饰器是 Python 中的一种通用结构，用于系统地修改既有函数。装饰器通常放在函数开头，并以 @ 字符打头。在本书中，我们将使用装饰器来创建设置函数和获取函数。

现在可以像下面这样编写代码：

```
>>> p = Person('Jen', 25)
>>> p
Person('Jen', 25)
>>> p.set_age(30)
>>> p
Person('Jen', 30)
>>> p.set_age(-6)
>>> p
Person('Jen', 30)
```

对于这种设置函数，一种常见的抱怨是，输入 p.set_age(30) 比输入 p.age = 30 更烦琐。为解决这种问题，可使用特性装饰器（property decorator）。

10.4.1　特性装饰器

特性装饰器融变量的简洁与函数的灵活于一身。装饰器指出函数或方法有点特殊，这里使用它们来指示设置函数和获取函数。

获取函数返回变量的值，我们将使用 @property 装饰器来指出这一点：

```
@property
def age(self):
  """ Returns this person's age.
  """
  return self._age
```

这个 age 方法除必不可少的 self 外不接受任何参数。我们在它前面加上了 @property，指出这是一个获取函数。这个方法的名称将被用于设置变量。

我们还将底层变量 self.age 重命名为 self._age。在对象变量前加上下划线是一种常见的做法，这里使用这种方式将这个变量与方法 age 区分开来。你需要将 Person 类中的每个 self.age 替换为 self._age。为保持一致性，最好也将 self.name 都替换为 self._name。修改后的 Person 类类似于下面这样：

```
# person.py
class Person:
  def __init__(self, name = '',
               age = 0):
    self._name = name
    self._age = age

  @property
  def age(self):
    return self._age

  def set_age(self, age):
    if 0 < age <= 150:
      self._age = age

  def display(self):
    print(self)

  def __str__(self):
    return "Person('%s', %s)" %
    → (self._name, self._age)

  def __repr__(self):
    return str(self)
```

为给 age 创建设置函数，我们将方法 set_age 重命名为 age，并使用 @age.setter 进行装饰：

```
@age.setter
def age(self, age):
  if 0 < age <= 150:
    self._age = age
```

完成这些修改后，就可以像下面这样编写代码了：

```
>>> p = Person('Lia', 33)
>>> p
Person('Lia', 33)
>>> p.age = 55
>>> p.age
55
>>> p.age = -4
>>> p.age
55
```

由于给 age 提供了设置函数和获取函数，编写的代码就像直接使用变量 age，但差别在于：遇到代码 p.age = -4 时，Python 实际上将调用方法 age(self, age)；同样，遇到代码 p.age 时，将调用方法 age(self)。这提供了如下优点：赋值语法很简单，同时可控制变量的设置和获取方式。

10.4.2　私有变量

依然可以直接访问 self._age：

```
>>> p._age = -44
>>> p
Person('Lia', -44)
```

问题在于，直接修改 _age 可能导致对象不一致，因此通常不希望直接修改的情况发生。

为降低变量 self._age 被直接修改的可能性，一种方式是将其重命名为 self.__age，即在变量名开头包含两个下划线。两个下划线表明 age 是私有变量，不应在 Person 类外直接访问它。要直接访问 self.__age，需要在前面加上 _Person，如下所示：

```
>>> p._Person__age = -44
>>> p
Person('Lia', -44)
```

这虽然不能禁止你直接修改内部变量，但将无意间这样做的可能性几乎降到了零。

术语说明

不以下划线打头的变量是公有变量，任何代码都可访问它们。

提示 编写大型程序时，一条实用的经验规则是，首先将所有对象变量都设置为私有的（即以两个下划线打头），再在有充分理由的情况下将其改为公有的。这可避免无意间修改对象内部变量导致的错误。

提示 乍一看，创建设置函数和获取函数的语法有点怪，但习惯后你就会发现这种语法非常清晰。别忘了，并非总是需要创建设置函数和获取函数；对于简单对象（如最初的 Person），使用常规变量可能很不错。

提示 有些程序员喜欢尽可能不提供设置函数，让对象是不可变的，就像数字、字符串和元组一样。如果对象没有设置函数，则创建它后，就无法通过"正规渠道"对其做任何修改。与其他不可变对象一样，这可避免众多微妙的错误，并让不同的变量共享同一个对象（从而节省内存）。当然这也存在缺点，那就是如果你需要修改对象，唯一的选择是创建一个体现了变更的新对象。

提示 如果程序员试图将 age 设置为指定范围外的值，age(self,age) 什么都不会做。另一种方法是故意引发异常，要求设置 age 的代码必须处理这种异常。引发异常的优点在于，这可能有助于发现其他错误。试图将 age 设置为荒谬的值可能表明程序的其他地方有问题。

10.5　继承

继承是一种重用类的机制，让你能够这样创建全新的类：给既有类的拷贝添加变量和方法。

假设我们要开发一款游戏，其中涉及人类玩家和计算机玩家。为此，可创建一个 Player 类，它包含所有玩家都有的东西，如得分和名称：

```
# players.py
class Player:
  def __init__(self, name):
    self._name = name
    self._score = 0
  def reset_score(self):
    self._score = 0
  def incr_score(self):
    self._score = self._score + 1
  def get_name(self):
    return self._name
  def __str__(self):
    return "name = '%s', score = %s"
    ➝ % (self._name, self._score)
  def __repr__(self):
    return 'Player(%s)' % str(self)
```

我们可以像下面这样使用 Player 对象：

```
>>> p = Player('Moe')
>>> p
Player(name = 'Moe', score = 0)
>>> p.incr_score()
>>> p
Player(name = 'Moe', score = 1)
>>> p.reset_score()
>>> p
Player(name = 'Moe', score = 0)
```

术语说明

用于描述继承的术语很多。如果 Human 类继承了 Player 类，我们可以这样说：

- Human 扩展了 Player。
- Human 从 Player 派生而来。
- Human 是 Player 的子类，而 Player 是 Human 的超类。
- Human 是一个 Player。

最后一个术语（是一个）意味着所有人都是玩家。要创建类层次结构，一种方式是考虑类之间可能存在的是一个关系。

咱们假设有两类玩家：人和计算机。主要差别在于，人通过键盘输入走法，而计算机使用函数生成走法。除此之外，这两类玩家相同，它们都有名称和得分。

下面来编写一个 Human 类，用于表示人类玩家。为此，一种办法是通过复制并粘贴新建 Player 类的一个拷贝，再添加让玩家走棋的方法 make_move(self)。这种办法虽然可行，但更佳的做法是使用继承。我们可以让 Human 类继承 Player 类的所有变量和方法，这样就不需要再次编写它们了：

```
class Human(Player):
  pass
```

在 Python 中，pass 语句表示“什么都不做”。对 Human 类来说，这是一个完整而实用的定义，它继承 Player 的代码，让我们能够像下面这样做：

```
>>> h = Human('Jerry')
>>> h
Player(name = 'Jerry', score = 0)
>>> h.incr_score()
>>> h
Player(name = 'Jerry', score = 1)
>>> h.reset_score()
>>> h
Player(name = 'Jerry', score = 0)
```

考虑到我们只为 Human 类编写两行代码，这相当令人难忘！

重写方法

一个小瑕疵是，h 的字符串表示为 Player，但更准确的说法应该是 Human。为修复这种问题，可给 Human 定义方法 __repr__：

```
class Human(Player):
  def __repr__(self):
    return 'Human(%s)' % str(self)
```

这样结果将如下：

```
>>> h = Human('Jerry')
>>> h
Human(name = 'Jerry', score = 0)
```

这就是方法重写：Human 中的方法 __repr__ 重写了从 Player 那里继承的方法 __repr__。这是定制派生类的常用方式。

现在，可轻松编写类似的 Computer 类，用于表示计算机玩家：

```
class Computer(Player):
  def __repr__(self):
    return 'Computer(%s)' % str(self)
```

这 3 个类组成了一个小型的类层次结构，如图 10-2 的类图所示。Player 为基类，而其他两个类为派生（扩展）类。

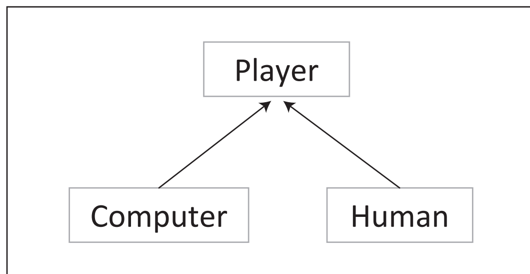


图 10-2　类图指出了 Player、Human 和 Computer 类之间的关系。其中的箭头表示继承，而整个类图是一个由类组成的层次结构。较抽象（通用）的类位于顶部附近，而较具体（特殊）的类位于底部附近 从本质上说，扩展类继承了基类的变量和方法；要让派生类共享的所有代码都应放在基类中

术语说明

基类常被称为**父类**，而派生类常被称为**子类**。

10.6　多态

为演示 OOP 的威力，咱们来创建一个名为 Undercut 的简单游戏。在这个游戏中，两个玩家同时选择一个 1~10 的整数，如果一个玩家选择的整数比对方选择整数的小 1，则该玩家获胜，否则算打平。例如，如果 Thomas 和 Bonnie 一起玩游戏 Undercut，且他们选择的数字分别为 9 和 10，则 Thomas 获胜；如果他们分别选择 4 和 7，则打成平手。

下面是一个玩 Undercut 游戏的函数：

```
def play_undercut(p1, p2):
  p1.reset_score()
  p2.reset_score()
  m1 = p1.get_move()
  m2 = p2.get_move()
  print("%s move: %s" % (p1.get_name(), m1))
  print("%s move: %s" % (p2.get_name(), m2))
  if m1 == m2 - 1:
    p1.incr_score()
    return p1, p2, '%s wins!' % p1.get_name()
  elif m2 == m1 - 1:
    p2.incr_score()
    return p1, p2, '%s wins!' % p2.get_name()
  else:
    return p1, p2, 'draw: no winner'
```

如果你仔细阅读这个函数的代码，将发现调用了 `p1.get_move()` 和 `p2.get_move()`。我们还没有实现这些函数，因为它们随游戏而异。下面就来实现这些函数。

10.6.1 实现 get_move 函数

虽然在游戏 Undercut 中，走法不过是选择 1~10 的数字，但人和计算机选择数字的方式截然不同。人类玩家通过键盘输入一个 1~10 的数字，而计算机玩家使用函数来选择数字。因此，Human 和 Computer 类需要专用的 get_move(self) 方法。

下面是 Human 类的方法 get_move（为节省篇幅，精简了错误消息，在网站的配套源代码中，包含更完整、更友好的消息）：

```
class Human(Player):
  def __repr__(self):
    return 'Human(%s)' % str(self)

  def get_move(self):
    while True:
      try:
        n = int(input('%s move (1 - 10): '
        → % self.get_name()))
        if 1 <= n <= 10:
          return n
        else:
          print('Oops!')
      except:
        print('Oops!')
```

上述代码不断要求用户输入一个 1~10 的整数，直到用户按要求做。try/except 结构用于捕获用户输入的不是整数（如 two）时函数 int 引发的异常。

对于计算机玩家，我们让它返回一个1~10的随机数（如果需要，以后可改进计算机采取的策略）：

```
import random


class Computer(Player):
  def __repr__(self):
    return 'Computer(%s)' % str(self)

  def get_move(self):
    return random.randint(1, 10)
```

10.6.2　玩游戏 Undercut

万事俱备，可以开始玩游戏 Undercut 了。先让人和计算机来玩这款游戏：

```
>>> c = Computer('Hal Bot')
>>> h = Human('Lia')
>>> play_undercut(c, h)
Lia move (1 - 10): 7
Hal Bot move: 10
Lia move: 7
(Computer(name = 'Hal Bot', score = 0),
➝ Human(name = 'Lia', score = 0),
➝ 'draw: no winner')
```

必须在函数 play_undercut 外面创建玩家对象，明白这一点很重要。这是一种良好的设计：函数 play_undercut 只关心玩游戏，而不关心如何初始化玩家对象。

函数 play_undercut 返回一个形式为 (p1, p2, message) 的三元组。其中 p1 和 p2 是传入的玩家对象；如果有玩家获胜，则将其得分加 1。message 是一个字符串，指出了获胜的玩家或打成平手。

也可以将两个计算机玩家传递给函数 play_undercut：

```
>>> c1 = Computer('Hal Bot')
>>> c2 = Computer('MCP Bot')
>>> play_undercut(c1, c2)
Hal Bot move: 8
MCP Bot move: 7
(Computer(name = 'Hal Bot', score = 0),
→ Computer(name = 'MCP Bot', score = 1),
→ 'MCP Bot wins!')
```

这次没有人类玩家，因此不会要求用户输入数字。

还可以传入两个人类玩家：

```
>>> h1 = Human('Bea')
>>> h2 = Human('Dee')
>>> play_undercut(h1, h2)
Bea move (1 - 10): 5
Dee move (1 - 10): 4
Bea move: 5
Dee move: 4
(Human(name = 'Bea', score = 0),
→ Human(name = 'Dee', score = 1),
→ 'Dee wins!')
```

通用接口

虽然传入两个 Human 对象时，play_undercut 也能正常运行，但这个接口不太合理：第二个玩家能够看到第一个玩家选择的数字！为让两个人玩这个游戏时比较有趣，你需要想办法避免第二个玩家看到第一个玩家选择的数字。

这 3 个示例（人对人、计算机对计算机和人对计算机）展示了**多态**的威力：使用相同的函数实现了截然不同的行为。我们没有编写 3 个不同的函数，而是只编写一个函数，并给它传递不同的对象。

实际上，这通常是一大优点。为使用多态，需要有一定的经验，还需特别注意设计细节，这需要更多时间和精力，但这样做通常物超所值。

10.7 更深入地学习

本章介绍了 OOP 的一些基本知识，Python 还有很多其他 OOP 功能，你可通过阅读在线文档进行学习。

一个重要主题是打造良好的面向对象设计。相比于仅仅使用对象，妥善地使用对象要难得多。为组织面向对象的程序，一种流行的方式是使用*面向对象设计模式*。面向对象设计模式是使用对象解决常见编程问题的处方，经过了实践的检验。

在探讨这个主题的著作中，Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 所著的《设计模式：可复用面向对象软件的基础》影响最为深远。掌握 OOP 的所有技术细节后，如果要了解重要的设计问题，阅读这部著作将是很不错的选择。

11

案例研究：文本统计

到目前为止，你见到的代码片段大多只有几行，旨在演示某项 Python 功能。编程新手会很快发现，将小小的代码片段组织成完整的程序是一大步。要编写规模较大的程序，必须更详尽地规划，还需对如何以最佳方式结合使用各项 Python 功能有所了解。刚开始编写较大的程序时，可能需要反复试验。

本章将循序渐进地开发一个较大的 Python 程序。首先对要解决的问题进行描述，然后创建一个解决问题的 Python 程序，并对其进行测试。

程序编写工作很棘手，但要演示这一点很难。看起来有了清晰的问题描述后，我们就找到了简洁的解决方案；但实际上绝不可能如此简单：需要反复试验，开始会遭遇失败，还常常需要推倒重来。通过编写程序，你将逐渐学会合并使用各种技术的最佳方式，并了解哪些解决方案通常对解决哪些问题行之有效。

本章内容

- 问题描述
- 保留想要的字母
- 使用大型数据文件测试代码
- 找出出现次数较多的单词
- 将字符串转换为次数字典
- 组织在一起
- 练习
- 最终的程序

11.1 问题描述

受邀为解决重要问题而编写程序时，编程新手常常不知道从何处着手。至少从笼统的角度说，答案很简单：编写大型程序时，先得明白要解决的问题。这看似简单，但未能正确认识要解决的问题是极其常见的编程错误。有时候，编写程序之所以很难，是因为你没有真正明白自己要做什么。

本章要解决的问题是，计算并打印有关文本文件内容的统计数据。我们想知道给定文本文件包含多少个字符、行和单词。除单词数外，我们还想知道在文件中出现次数最多的前 10 个单词，并按出现次数排列它们。

来看一个包含一小段文本的例子。

A long time ago, in a galaxy far, far away ...

它包含如下内容。

- 一行文本。我们假定换行字符（`\n`）被用于标识行尾，且不为空的文本文件至少包含一行。
- 46 个字符，包括空格和标点在内。
- 总共 10 个单词。然而，不同的单词只有 8 个，因为 `far` 和 `A` 都出现了两次。

在 Python 解释器中进行试验很有帮助，例如：

```
>>> s = 'A long time ago, in a galaxy far,
→ far away ...'
>>> len(s)
46
>>> s.split()
['A', 'long', 'time', 'ago,', 'in', 'a',
→ 'galaxy', 'far,', 'far', 'away', '...']
```

正如你看到的，函数 `len` 指出这个字符串包含 46 个字符；函数 `split` 将字符串划分为单词——如果忽略末尾的 `...`，字符串 `s` 总共包含 10 个单词。

如果仔细查看 `split` 返回的单词列表，将发现单词 `far` 出现了两次，但 `split` 将它们视为两个不同的字符串：“`far,`”（末尾有逗号）和“`far`”（没有逗号）。同样，`A` 和 `a` 也是相同的单词，只是大小写不同。

为处理上述细节，我们将精确地定义字符串为单词的含义：单词是包含一个或多个字符的字符串，其中每个字符都必须是小写字母 a~z。我们将忽略非字母字符（如数字和标点），并将大写字母转换为小写。因此前面的示例如下。

原始文本：A long time ago, in a galaxy far, far away ...

修改后的文本：a long time ago in a galaxy far far away

通过将修改后的句子划分为单词，得到的结果更准确：

```
>>> t = 'a long time ago in a galaxy
→ far far away'
>>> t.split()
['a', 'long', 'time', 'ago', 'in', 'a',
→ 'galaxy', 'far', 'far', 'away']
>>> len(t.split())
10
```

要计算不同的单词数，可将列表转换为集合（本书前面介绍过，集合不存储重复的值）：

```
>>> set(t.split())
{'a', 'ago', 'far', 'away', 'time', 'long',
→ 'in', 'galaxy'}
>>> len(set(t.split()))
8
```

删除非字母字符存在一些缺点。首先，字符数不对，因为有些字符被删除。但我们可采取这样应对的措施，即在修改前计算字符数。其次，对于有些单词，没有将其标点符号删除的好办法。例如，该如何处理 I'd 中的撇号呢？如果将其删除（并将 I 转换为小写），结果将为 id，与原来的单词不同。如果将撇号替换为空格，结果将为 I 和 d——一个是单词，一个不是。为解决这个问题，我们将撇号（还有连字符）视为字母。第三，改变大小写可能改变单词的含义。例如，将有些名字的首字母小写后，它将变成单词，如 Polish（polish） 和 Bonnie（bonnie）。 我 们不考虑这个问题，因为它看起来并非什么大问题。

11.2 保留想要的字母

接下来，考虑如何自动将字符串转换为所需的格式。将字符串转换为小写很容易，如下所示。

```
>>> s = "I'd like a copy!"
>>> s.lower()
"i'd like a copy!"
```

删除不想要的字符有点棘手，一种办法是使用字符串函数 replace 将不要的字符替换为空字符，例如：

```
>>> s = "I'd like a copy!"
>>> s.replace('!', '')
"I'd like a copy"
```

这种做法的问题在于，需要调用 replace 很多次：每种不需要的字符一次。相比于要保留的字符，要删除的字符多得多，因此这种做法的效率极低。

函数 normalize 的另一个版本

下面是实现函数 normalize 的更简洁方式：

```
def normalize2(s):
  """Convert s to normalized string.
  """
  return ''.join(c for c in s.lower()
  → if c in keep)
```

很多经验丰富的程序员都喜欢这个版本，因为它更简洁，且至少在他们看来易于理解。

正则表达式

解决这个问题的另一种方法是使用正则表达式。例如，可创建一个定义单词的正则表达式，再使用函数 findall 提取给定字符串中的所有单词。鉴于我们要显示基本的 Python 编程，因此不会在后续代码中使用任何正则表达式。

一种更佳的方法是保留想要的字母，例如：

```
\# 包含所有要保留的字符的集合
keep = {'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j',
  'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't',
  'u', 'v', 'w', 'x', 'y',
  'z',
  ' ', '-', "'"}

  def normalize(s):
  """Convert s to a normalized string.
  """
  result = ''
  for c in s.lower():
    if c in keep:
      result += c
  return result
```

这个函数以每次一个字符的方式遍历字符串 s，仅当字符包含在要保留的字符集合中时，才将其附加到 result 末尾。

11.3 使用大型数据文件测试代码

我们编写的代码不多，但足够做一些有用的实验。在下面的示例中，我们将使用文件 bill.txt，这是一个 5.4 MB 的文本文件，包含莎士比亚的全部作品，可从 Project Gutenberg 网站 www.gutenberg.org 免费下载。这个文件比较大，非常适合用于测试前述代码的效率。

要处理文本文件，方式之一是将整个文件作为一个字符串读取到内存中。下面在解释器中手工完成这项任务。

```
>>> bill = open('bill.txt', 'r').read()
>>> len(bill)
5465395
>>> bill.count('\n')
124796
>>> len(bill.split())
904087
>>> len(normalize(bill).split())
897610
```

从输出可知，这个文件包含大约 540 万个字符、大约 12.5 万行、大约 90 万个单词。

下面将所有代码放在一个函数中，以自动完成这项任务：

```
def file_stats(fname):
  """Print statistics for the given
  file.
  """
  s = open(fname, 'r').read()
  num_chars = len(s)
  num_lines = s.count('\n')
  num_words = len(normalize(s). split())

  print("The file '%s' has: " % fname)
  print("   %s characters" % num_chars)
  print("   %s lines" % num_lines)
  print("   %s words" % num_words)
```

调用函数 `file_stats` 得到的输出如下：

```
>>> file_stats('bill.txt')
The file 'bill.txt' has:
  5465395 characters
  124796 lines
  897610 words
```

在我的计算机上，运行这个函数需要大约 1 秒钟，这包括将文件加载到内存以及完成所有处理所需的时间。对一个简单的 Python 程序来说很不错！

11.4 找出出现次数较多的单词

来考虑如下问题：找出文本文件中出现次数较多的单词。这里的解决方案是，创建一个字典，其中的键为单词，值为单词在文件中出现的次数。

例如,对于前面的示例文本（经过规范化）:

a long time ago in a galaxy far far away

各个单词出现的次数如下：

```
     a: 2
  long: 1
  time: 1
   ago: 1
    in: 1
galaxy: 1
   far: 2
  away: 1
```

如果我们将上述内容转换为一个 Python 字典，结果将类似于下面这样：

```
d = {
         'a': 2,
      'long': 1,
      'time': 1,
       'ago': 1,
        'in': 1,
    'galaxy': 1,
       'far': 2,
      'away': 1
    }
```

可从这个字典提取很多有用的信息。

- d.keys() 是一个列表，包含文件中所有不同的单词。
- len(d.keys()) 是文件中不同的单词数。
- sum(d[k] for k in d) 是 d 中所有值之和，即文件包含的单词总数（包括重复的单词）。sum 是一个 Python 内置函数，返回序列的总和。

字典存储的数据未经排序，因此要获取一个清单，按出现次数从高到低的顺序列出所有单词，需要将字典转换为元组列表，如下所示。

```
lst = []
for k in d:
  pair = (d[k], k)
  lst.append(pair)
#
# [(2, 'a'), (1, 'ago'),
#  (1, 'galaxy'), (1, 'time'),
#  (2, 'far'), ...]
lst.sort()
#
# [(1, 'ago'), (1, 'away'),
#  (1, 'galaxy'), (1, 'in'),
#  (1, 'long'), ...]
lst.reverse()
#
# [(2, 'far'), (2, 'a'),
#  (1, 'time'), (1, 'long'),
#  (1, 'in'), ...]
```

其中的 for 循环将字典 d 转换为由元组 (count,word) 组成的列表。经过这样的转换后，就可使用列表函数 sort 按出现次数对数据排序。默认情况下，函数 sort 按从小到大的顺序排列数据，因为我们反转列表的排列顺序，将出现次数最多的单词（通常也是我们最感兴趣的单词）放在列表开头。

将 lst 中的单词按出现次数从高到低排列后，就可使用切片来获取出现次数最多的 3 个单词：

```
print(lst[:3])
#
# [(2, 'far'), (2, 'a'),
#  (1, 'time')]
```

如果要让输出更整洁，可以这样做：

```
for count, word in lst:
  print('%4s %s' % (count, word))
```

上述代码的输出如下：

```
2 far
2 a
1 time
1 long
1 in
1 galaxy
1 away
1 ago
```

注意到在每个单词的出现次数前面，都有 3 个空格。这是因为 print 语句包含格式命令 %4s，它让数字在宽度为 4 的字段中右对齐。只要没有单词出现的次数达到或超过 1000，这就可确保出现次数完全对齐。

11.5　将字符串转换为次数字典

下面来编写一个函数，它接受字符串 s 并生成一个字典，该字典的键为 s 中的单词，值为单词出现的次数：

```
def make_freq_dict(s):
  """Returns a dictionary whose keys
     are the words of s, and whose
     values are the counts of those
     words.
  """
  s = normalize(s)
  words = s.split()
  d = {}
  for w in words:
    if w in d:    \# 看到 w 出现过?
      d[w] += 1
    else:
      d[w] = 1
  return d
```

这个函数遍历字符串 s 的每个单词，并将其加入到字典中。如果 w 是包含在 d 中的键，if 语句 `if w in d` 的条件将为 True，否则为 False。如果 w 是包含在 d 中的键，则说明 w 出现过，因此将其出现次数加 1；如果 w 未包含在 d 中，就使用语句 `d[w] = 1` 将其作为新键加入到字典中。

11.6 组织在一起

现在万事俱备，可以编写一个函数，计算并显示给定文本文件的统计数据：

```
def print_file_stats(fname):
  """Print statistics for the given file.
  """
  s = open(fname, 'r').read()
  num_chars = len(s)               # 在规范化 s 之前计算字符数
  num_lines = s.count('\n')        # 在规范化 s 之前计算行数
  d = make_freq_dict(s)
  num_words = sum(d[w] for w in d)  # 计算 s 包含多少个单词
  # 创建一个列表，其中的元素由出现次数和单词组成的元组
  # 并按出现次数从高到低排列
  lst = [(d[w], w) for w in d]
  lst.sort()
  lst.reverse()
  # 在屏幕上打印结果
  print("The file '%s' has: " % fname)
  print("   %s characters" % num_chars)
  print("   %s lines" % num_lines)
  print("   %s words" % num_words)
  print("\nThe top 10 most frequent words are:")
  i = 1   # i 为列表元素编号
  for count, word in lst[:10]:
    print('%2s. %4s %s' % (i, count, word))
    i += 1
```

调用这个函数并将文件 bill.txt 传递给它，将得到如下打印输出：

```
The file 'bill.txt' has:
  5465395 characters
  124796 lines
  897610 words
The top 10 most frequent words are:
 1. 27568 the
 2. 26705 and
 3. 20115 i
 4. 19211 to
 5. 18263 of
 6. 14391 a
 7. 13606 you
 8. 12460 my
 9. 11107 that
10. 11001 in
```

这个单词清单完全在意料之中，虽然不那么激动人心。在英语文本中，出现频度最高的单词几乎都是很小的功能词，如 the 和 and。你必须往下找，才能找到更有趣的单词。

在我的计算机（典型的台式机）上，这个程序的执行时间不到 1.5 秒；考虑到文件如此之大，这样的结果相当不错。

11.7 练习

1. 修改函数 print_file_stats，使其也打印文件中不同的单词总数。

2. 修改函数 print_file_stats，使其打印文件中单词的平均长度。

3. 罕用语（hapax legomenon）是在文件中只出现过一次的单词。请修改函数 print_file_stats，使其打印罕用语总数。

4. 前面说过，文件 bill.txt 中出现频率最高的 10 个单词都是功能词，如 the 和 and。我们通常对这些单词不感兴趣，因此我们可创建一个排除词（stop word）集合，其中包含要忽略的所有单词。

在函数 print_file_stats 中新增一个名为 stop_words 的变量，如下所示：

```
stop_words = {'the', 'and', 'i', 'to', 'of',
→ 'a', 'you', 'my', 'that', 'in'}
```

当然，你可根据自己的喜好修改排除词集合。现在，修改程序的代码，在计算所有统计数据时，都将 stop_list 中的单词排除在外。

5.（较难）函数 `print_file_stats` 将一个文件名作为输入，并将整个文件都读取到一个字符串变量中。这种做法的问题在于，如果文件很大，将整个文件都放在一个字符串变量中将占用大量内存。

另一种做法是以每次一行的方式读取文件，这样占用的内存通常少得多。

请编写一个名为 `print_file_stats_lines` 的新函数，其功能与 `print_file_stats` 完全相同，但逐行读取输入文件。使用相同的文件调用这两个函数时，它们的输出应该相同。

11.8 最终的程序

最终的程序代码如下：

```
# wordstats.py
# 包含所有要保留的字符的集合
keep = {'a', 'b', 'c', 'd', 'e',
        'f', 'g', 'h', 'i', 'j',
        'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't',
        'u', 'v', 'w', 'x', 'y',
        'z',
        ' ', '-', "'"}

def normalize(s):
  """Convert s to a normalized string.
  """
  result = ''
  for c in s.lower():
    if c in keep:
      result += c
  return result

def make_freq_dict(s):
  """Returns a dictionary whose keys are the words of s, and whose values
  are the counts of those words.
  """
  s = normalize(s)
  words = s.split()
  d = {}
  for w in words:
    if w in d:  # 如果 w 出现过，就将其出现次数加 1
      d[w] += 1
    else:
      d[w] = 1  # 如果 w 是第一次出现，就将其出现次数设置为 1
  return d
```

```
def print_file_stats(fname):
  """Print statistics for the given file.
  """
  s = open(fname, 'r').read()

  num_chars = len(s)                  # 在规范化 s 之前计算字符数
  num_lines = s.count('\n')           # 在规范化 s 之前计算行数

  d = make_freq_dict(s)
  num_words = sum(d[w] for w in d)    # 计算 s 包含多少个单词

  # 创建一个列表，其中的元素由出现次数和单词组成的元组
  # 并按出现次数从高到低排列
  lst = [(d[w], w) for w in d]
  lst.sort()
  lst.reverse()

  # 在屏幕上打印结果
  print("The file '%s' has: " % fname)
  print("   %s characters" % num_chars)
  print("   %s lines" % num_lines)
  print("   %s words" % num_words)

  print("\nThe top 10 most frequent words are:")
  i = 1   # i 为列表元素编号
  for count, word in lst[:10]:
    print('%2s. %4s %s' % (i, count, word))
    i += 1

def main():
  print_file_stats('bill.txt')

if __name__ == '__main__':
  main()
```



深受欢迎的 Python 包

Python 深受欢迎的原因之一是有大量高品质的库，可帮助完成各种软件任务。附录 A 将介绍几个深受欢迎的包。

在这些库中，很多都只支持特定的 Python 版本（Python 可从 www.python.org 免费下载），牢记这一点很有帮助。具体地说，很多包还不支持 Python 3，因此要使用它们，您可能需要使用 Python 2.6（或更晚的版本）。所幸的是，如果您熟悉 Python 3，回过头去使用 Python 2 并不难。附录 B 简要地讨论了 Python 2 和 Python 3 之间的一些重大差别。

本章内容

- 一些深受欢迎的 Python 包

一些深受欢迎的 Python 包

PIL：Python 图像处理库

PIL（http://www.pythonware.com/products/pil/index.htm）是一个图像处理库，支持众多图像格式，可用于执行裁剪、大小调整、旋转和滤镜效果等操作。

Tkinter：Python GUI

Tkinter 是 Python 库自带的，是访问流行工具包 Tk GUI 的标准方式。如果您要使用 Python 创建图形用户界面（GUI），应首先考虑使用这个包。有关这个包的更详细信息，请参阅 http://docs.python.org/3/library/tkinter.html。

Django：交互式网站

Django（www.djangoproject.com）是一个用于创建交互式网站的框架。从这种意义上说，它类似于 Ruby on Rails，但使用的底层编程语言是 Python 而不是 Ruby。

Bottle：交互式网站

Bottle（http://bottlepy.org/docs/dev/）类似于 Django，因为它也是一个用于创建交互式网站的框架。不同的是，Bottle 是一个轻量级的小型框架，可能更适合用于开发小型网站。

Pygame：2D 动画

Pygame（www.pygame.org）让您能够创建和控制二维动画，尤其适合用于开发游戏。它提供了创建动画和声音的工具，还提供了控制游戏杆等输入设备的工具。Pygame 网站还提供了初步教程和示例程序，可帮助您快速上手。

SciPy：科学计算

SciPy（www.scipy.org）是一个用于科学计算的大型软件工具库，深受大家的欢迎，还有专门的会议！它提供的数学软件让您能够完成如下任务：求解最优化问题、执行线性代数数字计算、处理信号等。

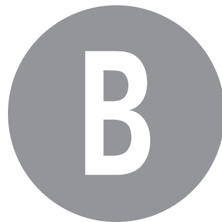
Twisted：网络编程

Twisted（http://twistedmatrix.com/trac）是一个深受欢迎的 Python 网络编程库，支持众多网络协议，可用于开发 Web 服务器、邮件服务器和聊天客户端 / 服务器等。

PyPI：Python 包索引

Python 包索引（http://pypi.python.org/pypi）是一个更新频繁的清单，列出了数千个用户提交的 Python 包。如果要寻找专用 Python 库或了解 Python 已用于哪些方面，这是一个不错的地方。

在网上搜索可轻松找到数千个其他的 Python 库。无论是什么编程任务，只要有人做过，几乎都有相关的 Python 库！



比较 Python 2 和 Python 3

Python 3 发布于 2008 年底，是一次重大的 Python 升级。Python 3 的有些改进不向后与 Python 2 兼容，因此 Python 2 始终与 Python 3 并行地向前发展。

附录 B 概述 Python 3 的一些重大改进，阐述如何将 Python 2 程序转换为 Python 3 程序。

本章内容

- ❑ Python 3 新增功能

Python 3 新增功能

Python 3 新增了很多功能，一些最显著的新功能如下。

- 在 Python 3 中，print 是货真价实的函数；而在 Python 2 中，print 是个语言结构，像 if 和 for 一样。在 Python 2 中，print 存在的问题是难以修改，例如，在 Python 3 中，修改 print 语句使其打印到文件而不是控制台要容易得多，因为只需给函数 print 重新赋值即可。

- 在 Python 3 中，整数除法的结果完全符合预期：

  ```
  Python3>>> 1 / 2
  0.5
  ```

  然而，在 Python 2 中执行整数除法时，将删除小数部分：

  ```
  Python2>>> 1 / 2
  0
  ```

  虽然 Python 2 的整数除法方式也得到了其他编程语言的支持，但很多程序员都认为它有悖于直觉，可能导致微妙的错误。

- Python 2 有两种类：老式类和新式类，而 Python 3 完全抛弃了老式类。

- Python 3 重命名了两个重要的函数：函数 input 和 range 在 Python 2 中分别名为 raw_input 和 xrange。

- 第 9 章介绍的格式字符串只有 Python 3 支持，Python 2 不支持。Python 2 只支持使用运算符 % 的字符串插入。

在技术方面，Python 3 还做了众多其他的改进。要全面了解 Python 3 和 Python 2 之间的差异，请参阅 What's New in Python 3.0（http://docs.python.org/3/whatsnew/3.0.html）。

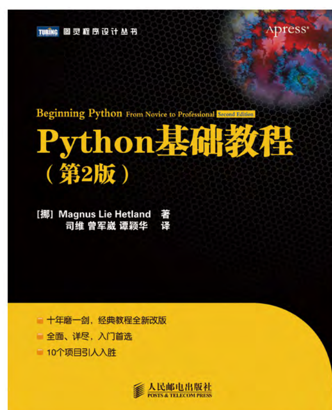
将 Python 2 程序转换为 Python 3 程序通常不难，有一个工具可助一臂之力，它就是 2to3（http://docs.python.org/3/library/2to3.html），几乎能够将任何 Python 2 程序自动转换为等价的 Python 3 程序。

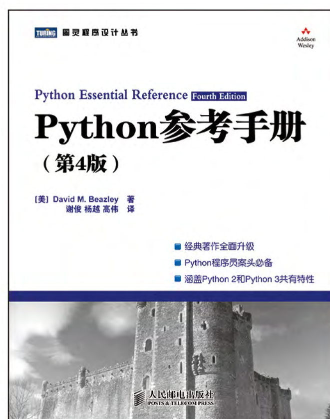
该使用哪个 Python 版本

决定使用哪个 Python 版本（Python 2 还是 Python 3）时，需要考虑以下几个因素。

- 如果必须使用 Python 2 程序，可能应选择 Python 2；否则，就得将既有的 Python 2 程序转换为 Python 3 程序，而这可能很难。
- 有些专用库只支持某个 Python 版本，如果需要使用这样的库，在选择 Python 版本方面可能受到限制。
- 如果你刚从事编程工作，不用维护老式 Python 程序，也无需使用专用库，那么使用 Python 3 可能是最佳选择。

索　　引


图灵程序设计丛书
Apress
Beginning Python From Novice to Professional Second Edition
Python基础教程
（第2版）
[挪] Magnus Lie Hetland 著
司维 曾军崴 谭颖华 译
十年磨一剑，经典教程全新改版
全面、详尽、入门首选
10个项目引人入胜
人民邮电出版社
POSTS & TELECOM PRESS


图灵程序设计丛书
Addison Wesley
Python Essential Reference Fourth Edition
Python参考手册
（第4版）
[美] David M. Beazley 著
谢俊 杨越 高伟 译
经典著作全面升级
Python程序员案头必备
涵盖Python 2和Python 3共有特性
人民邮电出版社
POSTS & TELECOM PRESS


图灵程序设计丛书
真实项目中的经验总结
行业精英们的智慧结晶
Python
开发实战
人民邮电出版社
POSTS & TELECOM PRESS

Python编程入门（第3版）

Python是一种解释型、面向对象、动态数据类型的高级程序设计语言，自20世纪90年代初诞生至今，逐渐被广泛应用于处理系统管理任务和Web编程。

本书用通俗易懂的语言结合常见任务、屏幕图和详细的解释，循序渐进地介绍了Python的基础知识，助你轻松、迅速地学习Python。书中没有深奥的理论或者高级应用，非常适合用来自学。读罢本书，你定能掌握Python的各项基础知识，成为一名真正的Python程序员！

本书特色：

- 简洁的步骤和解释 助你快速上手。
- 必备的参考指南 值得反复品味。
- 学习Python基础知识的不二选择 从算术运算、字符串、变量，到函数、数据结构、输入输出和异常处理，应有尽有！
- 配套网站 https://code.google.com/p/pythonintro/上提供了示例程序及其他材料。





图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/软件开发

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-33374-2
定价：39.00元

欢迎加入

图灵社区

电子书发售平台

电子出版的时代已经来临，在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为"敏捷出版"，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书"出版即过时"的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版的梦想。你可以联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者，这极大地降低了出版的门槛。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果有意翻译哪本图书，欢迎来社区申请。只要通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

读者交流平台

在图灵社区，读者可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。欢迎大家积极参与社区开展的访谈、审读、评选等多种活动，赢取银子，可以换书哦！